Clube Delphi

Arquitetura: MVVM
Uma implementação para desenvolvedores Delphi

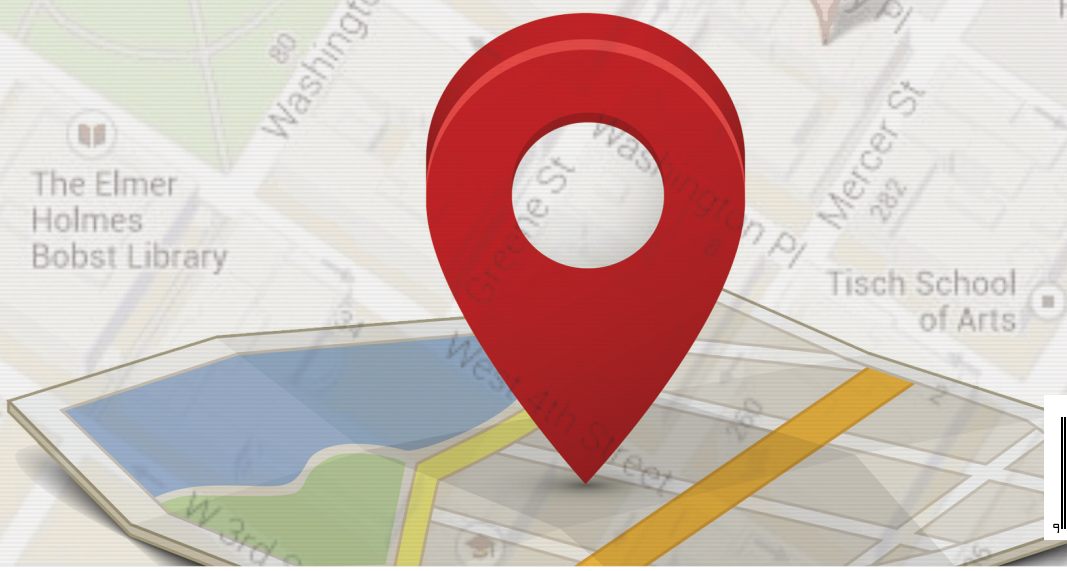
Princípios SOLID - Parte 2
A solidez que seu sistema precisa

Edição 161 :: R$ 14,90

DEVMEDIA

GOOGLE MAPS E DELPHI

Use a API, crie rotas e muito mais



ISSN 1517990-7

Analisando a linguagem Delphi
Características que fazem a diferença

Controle total de usuários
Construa uma solução personalizada

SEU CARRO TEM SEGURO, SUA SAÚDE TEM SEGURO, MAS E O SEU EMPREGO... TÁ SEGURO??

NÃO DEIXE JUSTAMENTE A SUA CARREIRA FICAR EM RISCO!

Manter-se atualizado com todas as novidades do mercado de desenvolvimento é obrigação de todo bom programador. Faça agora mesmo um seguro para a sua carreira. Seja um assinante MVP!

Saia do risco!

TENHA ACESSO A:

- +DE 290 CURSOS ONLINE
- 09 REVISTAS MENSAIS
- 9.074 VÍDEO-AULAS

POR APENAS 69,90 MENSAIS

ASSINATURA MVP MOST VALUABLE PROFESSIONAL

QUEM TEM ESTÁ TRANQUILO.

DEVMEDIA | Acesse: www.devmedia.com.br/mvp

Clube Delphi

161ª Edição 2014 ISSN 1517990-7 Impresso no Brasil

Editor Geral

Paulo Quicoli (pauloquicoli@gmail.com)

Equipe Editorial

Guinther Pauli (guintherpauli@gmail.com)

Fabrício Hissao Kawata (fabricio.kawata@bol.com.br)

Giuliano Scombatti Pinto (giuliano@sygnux.com.br)

Daniel Sobrinho Laporte (daniel.laporte@gmail.com)

Jornalista Responsável

Kaline Dolabella - JP24185

Consultor Técnico

Joel Neto (joelrlneto@gmail.com)

Capa e Diagramação

Romulo Araujo

Atendimento ao Leitor

A DevMedia conta com um departamento exclusivo para o atendimento ao leitor. Se você tiver algum problema no recebimento do seu exemplar ou precisar de algum esclarecimento sobre assinaturas, exemplares anteriores, endereço de bancas de jornal, entre outros, entre em contato com:

www.devmedia.com.br/central
(21) 3382-5038

Publicidade

Para informações sobre veiculação de anúncio na revista ou no site e para fechar parcerias ou ações específicas de marketing com a DevMedia, entre em contato com:

publicidade@devmedia.com.br

Distribuição

FC Comercial e Distribuidora S.A
Rua Teodoro da Silva, 907
Grajaú - RJ - 206563-900

Fale com o Editor!

É muito importante para a equipe saber o que você está achando da revista: que tipo de artigo você gostaria de ler, que artigo você mais gostou e qual artigo você menos gostou. Fique a vontade para entrar em contato com os editores e dar a sua sugestão!

Se você estiver interessado em publicar um artigo na revista ou no site ClubeDelphi, entre em contato com os editores, informando o título e mini-resumo do tema que você gostaria de publicar:

Paulo Quicoli - Editor da Revista
pauloquicoli@gmail.com

Paulo Quicoli - **Editor Geral**

pauloquicoli@gmail.com
quicoli.wordpress.com
twitter.com/pauloquicoli

MVP

Assine agora e tenha acesso a todo o conteúdo da DevMedia: **www.devmedia.com.br/mvp**

DEVMEDIA

Sumário

Conteúdo sobre Boas Práticas, Artigo no estilo Curso

04 – Princípios SOLID com Delphi – Parte 2
[*Filipe Dalepiane*]

Artigo no estilo Solução Completa

14 – Google Maps: criando mapas com Delphi para Android
[*Rafael Mota*]

Artigo no estilo Solução Completa

21 – Controles de permissão customizados no Delphi
[*Daniel Sobrinho Laporte*]

Conteúdo sobre Arquitetura

34 – MVVM para programadores Delphi
[*Fabricio Hissao Kawata*]

Artigo no estilo Engenharia de Software

42 – Entenda a Delphi Language
[*Filipe Dalepiane*]

Dê seu feedback sobre esta edição!

A ClubeDelphi tem que ser feita ao seu gosto. Para isso, precisamos saber o que você, leitor, acha da revista!

Dê seu voto sobre esta edição, artigo por artigo, através do link:

www.devmedia.com.br/clubedelphi/feedback

Princípios SOLID com Delphi – Parte 2

Aprenda a criar suas classes obedecendo estes importantes princípios da orientação a objetos

Este artigo faz parte de um curso

Fique por Dentro

Neste artigo temos a segunda parte da série princípios SOLID da orientação a objetos. Nesta segunda parte faremos um estudo dos dois últimos princípios que são o princípio da segregação de interfaces e o princípio da inversão de dependência. Além disso, veremos algumas métricas de análise de softwares com baixa qualidade e que precisam de refatoração para melhorar seu design.

Como sabemos, a mudança de requisitos é considerado por muita gente como o principal fator de degradação de um software, mas na realidade não são, porque softwares baseados em princípios e padrões de desenvolvimento devem ser propensos a estas mudanças, sem que as mesmas afetem a arquitetura do sistema.

A realidade é que a mudança de requisitos é uma das grandes "desculpas" dadas pelos desenvolvedores, principalmente aqueles iniciantes que ainda não tem uma formação sólida e conhecimento aprofundado de arquitetura de sistemas. Este tipo de pensamento também vem de uma cultura existente no desenvolvimento de sistemas, onde se tenta de todas as maneiras evitar a alteração em um software, na maioria das vezes em decorrência e conhecimento da fragilidade de determinado sistema, pois se sabe que ele não está preparado para mudanças.

As metodologias ágeis surgem como a solução para muitos destes problemas, visto que um dos princípios dela é a de que um software obrigatoriamente terá mudanças, ficando a responsabilidade para o desenvolvedor adequar o software já no início do desenvolvimento para que o mesmo permaneça com um bom design mesmo depois de seguidas modificações.

Outra característica presente nas metodologias ágeis é que o desenvolvimento de um software deve ser feito de forma incremental, conforme os desenvolvedores vão entendendo as regras de negócio, sendo que a cada nova iteração, o design do sistema deve ser mantido de acordo com os requisitos e sempre com um código claro e limpo.

Quando surge a necessidade de uma modificação num sistema existente, como a correção de algum erro ou a adição de uma nova funcionalidade, não devemos encarar isso como um problema, e sim como uma oportunidade de qualificarmos o software e disponibilizar para o usuário uma ferramenta eficiente.

Indícios de software de baixa qualidade

Existem diversas formas de diagnosticarmos softwares com baixa qualidade, seguem algumas delas:

• **Rigidez**: Um software dito rígido é um software muito difícil de ser modificado/alterado. Uma simples mudança ocasiona uma série de mudanças em cascata em várias outras partes do sistema, de forma que além do trabalho de darmos manutenção nestas outras partes do sistema, teremos também que realizar todos os testes nestas unidades que foram alteradas. Veja que uma alteração em uma parte do sistema fez com que precisássemos alterar outras que estavam funcionando perfeitamente, de maneira que podemos ter adicionado bugs nestas outras partes do sistema, sendo que a mesma não tem nada a ver com a alteração original. Isso é um sinal de mal design de software.

• **Fragilidade**: Temos um software frágil quando as mais simples mudanças acabam gerando diversos erros em vários outros módulos do software. Na maioria das vezes os erros ocorrem em módulos que não tem ou não deveria ter nenhuma relação com o módulo original. Um indício de fragilidade do software é quando os desenvolvedores tem muito receio em alterar determinada parte daquele software, com medo de esta alteração "estourar"

em outras partes do sistema. Na maioria das vezes os desenvolvedores, as vezes por falta de conhecimento ou até por pressão de superiores para a rápida manutenção acabam não fazendo todos os testes necessários no módulo alterado e todos os seus dependentes, sendo isso um grave risco de apenas o usuário final se deparar com os erros em ambiente de produção.

• **Imobilidade**: A imobilidade significa a dificuldade de reaproveitamento de determinados módulos do sistema em outros sistemas, ou seja, a dificuldade em utilizarmos classes que deveriam ser iguais nos dois sistemas, mas devido a falhas no desenvolvimento, essas classes foram aumentando e assumindo responsabilidades extras que não cabiam as mesmas, impedindo que outro software pudesses a utilizar, fazendo com que diminuísse também o tempo de desenvolvimento do software final. Geralmente há um grande medo por parte dos desenvolvedores na possibilidade de reaproveita aquele módulo, pois ao olhar para o código, as vezes já percebemos que as classes estão mal projetadas.

• **Viscosidade**: A viscosidade de um software está relacionada com a melhor forma de fazer manutenção do mesmo, ou seja, para realizarmos uma mudança no software é muito mais fácil fazermos uma "gambiarra" para esta modificação do que desenvolvermos um código mais conciso e mantermos o bom design, ou seja, é muito mais fácil alterar um software da maneira errada do que da maneira certa. Isso ocorre muito também com desenvolvedores menos experientes, que desconhecem boas práticas de programação e padrões de projeto, de forma que acabam prejudicando o design da classe e fazendo com que o software perca qualidade.

• **Alta Complexidade**: Um software que possui muitas unidades desnecessárias, chamamos isso também de excesso de engenharia, quando o desenvolvedor prevê muitas mudanças futuras e aplica as vezes muitos padrões de projetos, aumentando consideravelmente a complexidade do software de forma desnecessária, na expectativa de atender demandas futuras, prevendo diversas situações que podem ocorrer no futuro, situações estas que na maioria das vezes nem vão ocorrer. Isso torna o código complexo demais dificultando muito o entendimento, principalmente se alguma manutenção for dada por outro desenvolvedor que não seja o criador original desta classe. Existem dois princípios menos conhecidos que servem para evitar esta alta complexidade criada de forma errônea, são eles o KISS e o YAGNI (**BOX 1**).

• **Redundância**: Com certeza essa é a mais famosa, a duplicidade de código em várias partes da aplicação. Isso acontece porque os desenvolvedores ao invés de encapsular uma determinada ação em um método ou classe, preferem simplesmente copiar e colar o código já existente, causando redundância. Um exemplo clássico que poderíamos citar é o da validação de CPF, já existe um método em uma classe chamado ValidarCpf, porém este método agora precisa ser utilizado em outra classe que não tem relação nenhuma com esta, então ao invés do desenvolvedor refatorar esta classe criando uma nova classe chamada TValidaCpf com um método ValidarCpf para que ambas possam utilizá-la, mas ele simplesmente copia o método e cola na nova classe causando a repetição deste código, e pior, se um dia precisarmos ajustar esta validação precisaremos lembrar de modificar a implementação em ambas as classes.

• **Código de Difícil Leitura**: Esta é uma das maiores dificuldades encontradas para desenvolvedores que devem dar manutenção em softwares, pela dificuldade de entendimento do código escrito, por ser desorganizado ou por nomes de métodos e variáveis sem sentido, ou até nomes de classes e interfaces que não expressam no seu nome a sua clara responsabilidade.

BOX 1. KISS e YAGNI

KISS (Keep it Simple) ou ainda Mantenha Simples, é um princípio de desenvolvimento de softwares que valoriza a simplicidade do projeto e defende que toda a complexidade desnecessária seja descartada.

YAGNI (You aren't gonna need it) ou ainda você ainda não vai precisar dele, é um princípio que sugere que os desenvolvedores não adicionem funcionalidades ao software sem que estas necessidades sejam realmente necessárias, ou seja, não se deve prever o que será preciso antes de ser realmente necessário.

KISS e YAGNI são dois princípios que demonstram uma prática de XP – Extreme Programming, que pregam o desenvolvimento ágil baseado em refatorações. Não se devem aplicar padrões de projeto de software antecipadamente, pois causa complexidade desnecessária. Quando se inicia um projeto de software não conseguimos ter a visão de como a arquitetura irá evoluir e quais padrões de projeto serão necessários, se é que um dia precisarão ser aplicados.

SOLID

Os princípios SOLID são um conjunto de cinco princípios básicos de boas práticas de programação para o desenvolvimento de aplicações orientadas a objetos e de essencial conhecimento para qualquer desenvolvedor que deseje desenvolver sistemas de fácil manutenção, com alta escalabilidade e com códigos limpos e elegantes. Estes princípios farão com que nossas classes possuam um melhor design. São eles:

• SRP: Single Responsibility Principle: princípio da responsabilidade única;

• OCP: Open Closed Principle: princípio do aberto/fechado;

• LSP: Liskov Substituition Principle: princípio da substituição de Liskov;

• ISP: Inteface Segregation Principle: princípio da segregação de interfaces;

• DIP – Dependency Inversion Principle: princípio da inversão de dependência.

ISP - Inteface Segregation Principle

O Princípio da Segregação de Interfaces (ISP - Interface Segregation Principle) nos ensina que numa aplicação que segue o paradigma orientado a objetos, é muito melhor termos várias interfaces específicas do que uma interface de uso geral. Uma classe cliente nunca ser dependente de métodos que não utiliza.

Segregar significa separar, ou seja, devemos pegar nossa interface e separá-la em partes mais lógicas. Então devemos dividir nossas interfaces grandes, com muitos métodos e muitas responsabilidades em interfaces menores, para que as classes

clientes dependam apenas de métodos que realmente irão utilizar e não de métodos que não farão sentido para ela.

Quando desenvolvemos interfaces genéricas demais, acabamos fazendo com que uma implementação (classes concretas), não utilizem alguns métodos da interface. Quando isso ocorre simplesmente precisamos implementar métodos que não fazem nada, somente estão ali para cumprir a regra de implementação de interfaces, que diz que quando implementamos uma interface todos os métodos presentes nela devem ser obrigatoriamente implementados.

Se seguirmos fielmente este princípio da orientação a objetos nosso software será mais escalável, terá uma manutenção facilitada. Como regra geral, sempre que precisarmos de métodos novos em uma interface, veja se este método realmente faz sentido nesta interface, se não fizer, deve-se criar uma nova interface para ele.

Temos o princípio da responsabilidade única relacionado também a este princípio, porque se temos uma interface com muitos métodos, há de ser supor que esta interface não está seguindo o princípio da responsabilidade única e provavelmente deve estar realizando mais de uma tarefa, ferindo o princípio.

Por fim, uma dica é que sempre devemos ter nossas interfaces enxutas, com poucas ações ou métodos. Interfaces com muitos métodos prejudicam muito a manutenção, visto que ao modificarmos a interface, devemos percorrer por todos os pontos do sistema onde ela é utilizada e fazer as adequações necessárias. Estes cuidados melhoram o design das classes, ficando cada uma com apenas as suas responsabilidades, atendendo também o primeiro princípio SOLID que é o da responsabilidade única, aumentando bastante o nível de coesão (**BOX 2**) delas.

BOX 2. Coesão

A medida da Coesão de uma classe segundo a Orientação a Objetos, indica de que forma foi projetada uma determinada classes. Nos mostra como a classe possui uma funcionalidade única e clara. Desta forma quanto mais simples e focada for a classe para resolver um único problema, maior a sua coesão, algo desejado no desenvolvimento orientado a objetos. Classes com alta coesão trazem benefícios como a facilidade de manutenção e evolução, sendo também possível e com facilidade serem reutilizadas posteriormente em outros projetos.

DIP – Dependency Inversion Principle

Este princípio de Inversão de Dependência sem dúvida é um dos mais importantes da orientação a objetos, sendo que o mesmo é bastante estudado e existe uma série de frameworks nas mais variadas linguagens de programação para atendê-lo.

O princípio nos ensina que módulos de níveis mais alto do software nunca devem depender de módulos mais baixos, sempre deve depender de abstrações, estas abstrações também não devem depender de detalhes e sim os detalhes das abstrações.

Ao conseguirmos inverter a dependência, fazemos com que as classes clientes não fiquem frágeis em relação detalhes de implementação, ou seja, se fizermos alguma mudança na implementação nas classes que a classe cliente é dependente, não se faz necessário alterações na classe cliente.

Existem diversos padrões de projetos (**BOX 3**) relacionados com este princípio, pois na maioria das vezes desejamos criar interfaces que não dependam de implementações.

BOX 3. Padrões de Projeto GoF

Os padrões de projeto do GoF surgiram com o objetivo de padronizar a solução de problemas frequentes no desenvolvimento de um software orientado a objetos, para que estes padrões sejam utilizados na sua resolução, tornando-se estas resoluções um padrão. Receberam este nome GoF (Gang of Four) devido a publicação de um livro no ano de 1995 chamado Design Patterns: Elements of Reusable Object-Oriented Software por Erich Gamma, Richard Helm, Ralph Johnson e John Vlissides, a chamada Gangue of Four.

Padrões de criação: relacionados à criação de objetos (Abstract Factory, Builder, Factory Method, Prototype, Singleton);

Padrões estruturais: tratam das associações entre classes e objetos (Adapter, Bridge, Composite, Decorator, Facade, Flyweight, Proxy);

Padrões comportamentais: tratam das interações e divisões de responsabilidades entre as classes ou objetos (Chain of Responsibility, Command, Interpreter, Iterator, Mediator, Memento, Observer, State, Strategy, Template Method, Visitor).

Outro padrão que anda junto com a inversão de controle é o da injeção de dependências, pois na realidade a injeção de dependências é uma das maneiras de conseguirmos implementar a inversão de controle. O padrão de injeção de dependências se baseia em um padrão de projeto chamado Builder (**BOX 4**).

BOX 4. Builder

O padrão de projeto de software Builder permite a separação da construção de um objeto complexo da sua representação, de forma que o mesmo processo de construção possa criar diferentes representações. Segundo o livro Design Patterns: Elements of Reusable Object-Oriented Software, este padrão contém os seguintes elementos:

director — constrói um objeto utilizando a interface do builder;

builder — especifica uma interface para um construtor de partes do objeto-produto;

concrete builder — define uma implementação da interface builder, mantém a representação que cria e fornece interface para recuperação do produto;

product — o objeto complexo acabado de construir. Inclui classes que definem as partes constituintes.

Temos hoje no mercado diversos frameworks de injeção de dependências, principalmente para as linguagens Java e C#, onde o desenvolvimento orientado a objetos é obrigatório. Já no Delphi existem algumas iniciativas como o delphi-spring-framework e o delphiDIcontainer. Ao utilizarmos esses frameworks, as dependências entre as classes não são feitas explicitas programaticamente, e sim na infraestrutura do framework, também chamado de container, que fica sendo responsável por injetar as dependências da classe no momento apropriado. Neste sentido, as dependências são configuradas através do uso de XML ou através de reflexão computacional com o uso de atributos e anotações.

Este padrão prega que as dependências de nossas classes não devem ser criadas de maneira explicita e sim ser injetada de forma programática ou através de um framework. Podemos

exemplificar, por exemplo, uma classe DAO (TClienteDAO), ela depende de uma classe de conexão com a base de dados para efetuar operações de consulta e atualização na base de dados. Porém essa dependência com a classe de conexão com o banco de dados (TConexao) será injetada na classe TClienteDAO, de maneira que a classe DAO não precisa preocupar-se com o ciclo de vida do objeto de conexão (criar, destruir, etc.), ela apenas recebe esta instância e a usa.

Existem basicamente quatro maneiras de realizarmos a injeção de dependências em uma classe, são elas:

• Construtor: todas as dependências de uma classe são injetadas via método construtor da classe;

• Propriedade: as dependências da classe são injetadas através de propriedades (setter);

• Interface: a dependência que será injetada é uma abstração (interface ou classe abstrata) da classe concreta;

• Framework: frameworks facilitam a injeção de dependências, utilizando recursos extras como XML e reflexão computacional e armazenando as dependências em um container.

Por fim, podemos concluir que este princípio principalmente nos ensina que nossas classes devem depender sempre de abstrações, nunca de classes concretas, pois as abstrações (interfaces e classes abstratas) mudam com bem menos frequência, facilitando bastante à manutenção e evolução do software. Isso vai de encontro com um dos maiores lemas da orientação a objetos que diz: programe sempre voltado para a interface e nunca para a implementação. Com isso conseguimos diminuir o acoplamento entre as classes do sistema.

Princípio da Segregação de Interfaces na Prática

Neste exemplo faremos uso de um domínio bastante presente na vida de qualquer desenvolvedor, que é o da persistência. Na **Listagem 1** temos a declaração de uma classe TDAO que servirá de classe base para todas as classes de persistência, que deve herdá-la quando assim desejar.

Observe que além dos métodos de inserção/atualização (Salvar), temos alguns métodos de busca que deverão ser implementados por todas as classes filhas. Além desses, temos uma função de efetuar log das operações realizadas.

Veja a classe TFuncionarioDAO, ela herda de TDAO (linha 14) e implementa dos os métodos. No caso desta classe faz sentido a implementação de um log, já que geralmente esta classe possui atributos como salário, que precisam ter maior nível de segurança ao efetuar modificações. As buscas também fazem sentido, já que numa busca por funcionário podemos ter a busca por sua matrícula na instituição e pelo nome do mesmo.

Porém, a implementação do método BuscarTodos na classe TFuncionario não faz sentido, geralmente esse tipo de busca sobrecarrega o servidor, já que muitos dados devem ser trafegados na rede. Neste sentido, a busca por todos os funcionários não faz sentido e mesmo assim devemos implementá-la, mesmo que não venhamos a utilizar este método.

Agora veja uma classe de persistência ainda mais simples, a TUnidadeMedidaDAO (linha 25), esta classe apenas armazena um identificador e a descrição da sigla, de maneira que buscar por código e nome são completamente desnecessárias, visto que numa aplicação real, geralmente listamos todas as unidades de medida através do BuscarTodos. Nesse sentido os métodos BuscaPorCodigo e BuscaPorNome ficam sobrando nesta classe.

A funcionalidade de EfetuarLog também é totalmente desnecessária no contexto da tabela de Unidades de Medidas, visto que é uma tabela auxiliar e não demanda tanta segurança nas informações. Mesmo assim a classe TUnidadeMedida herda o atributo FArquivoLog e também é obrigada a implementar método EfetuarLog, do contrário não conseguimos compilar o programa.

Como pudemos observar na **Listagem 2**, temos um design que fere gravemente o princípio de segregação de interfaces, pois tanto na classe TFuncionarioDAO quanto na TUnidadeMedidaDAO são obrigados a implementar métodos que não irão fazer uso. Temos uma interface muito genérica, que foi criada para atender a vários requisitos. Devemos segregá-la (separá-la) em duas ou mais interfaces de modo que viemos a atender o princípio.

Se observarmos a nova implementação deste modelo, conforme a **Listagem 2**, veremos que foram feitas diversas refatorações no design das classes.

O primeiro detalhe a se observar é a transformação da classe abstract TDAO em uma interface IDAO (a partir da linha 1), já que esta classe não possuía implementação, sendo uma grande candidata a esta transformação em interface. Agora nossos métodos não precisam mais das diretivas virtual e abstract, porque todos os métodos de uma interface são assim sem a necessidade de diretivas. Além disso observe que somente ficaram os métodos que serão de uso de todas as possíveis classes filhas de IDAO, como o Salvar, Excluir e BuscarPorId. Os demais métodos foram extraídos da classe e deram origem a outras interfaces.

Os métodos de busca que diferem de uma classe DAO para outra foram separados em três novas interfaces IBuscaTodos (linha 8), IBuscaPorCodigo (linha 13) e IBuscaPorNome (linha 18), cada uma com um método a ser implementado e que será responsável pela busca.

Como não serão todas as classes que farão uso do log de alterações no banco de dados, faz sentido que também o atributo e o método que existiam anteriormente em uma nova interface chamada ILog (linha 23). Veja que não podemos declarar atributos numa interface, mas propriedades sim, de modo que precisamos obrigatoriamente definir métodos Get e Set para modificá-la (linhas 25 a 28).

Veja agora como ficou a classe TFuncionarioDAO, fazemos com que a mesma herde de TInterfacedObject que é uma das classes responsáveis por implementar a interface IInterface, que é a interface base para todas as demais interfaces do delphi, com os

Listagem 1. Classes ferindo o princípio da Segregação de Interfaces

```
1. TDAO = class abstract
2. protected
3.   FArquivoLog: string;
4. public
5.   procedure Salvar(AObject: TObject); virtual; abstract;
6.   procedure Excluir(AId: Integer); virtual; abstract;
7.   function BuscarPorId(AId: Integer): TObject; virtual; abstract;
8.   function BuscarTodos: TObjectList<TObject>; virtual; abstract;
9.   function BuscarPorCodigo(ACodigo: Integer): TObject;
     virtual; abstract;
10.  function BuscarPorNome(ANome: string): TObjectList<TObject>;
     virtual; abstract;
11.  procedure EfetuarLog; virtual; abstract;
12. end;
13.
14. TFuncionarioDAO = class(TDAO)
15. public
16.  procedure Salvar(AObject: TObject); override;
17.  procedure Excluir(AId: Integer); override;
18.  function BuscarPorId(AId: Integer): TObject; override;
19.  function BuscarPorCodigo(ACodigo: Integer): TObject; override;
20.  function BuscarPorNome(ANome: string): TObjectList<TObject>; override;
21.  procedure EfetuarLog; override;
22.  function BuscarTodos: TObjectList<TObject>; override;
23. end;
24.
25. TUnidadeMedidaDAO = class(TDAO)
26. public
27.  procedure Salvar(AObject: TObject); override;
28.  procedure Excluir(AId: Integer); override;
29.  function BuscarPorId(AId: Integer): TObject; override;
30.  function BuscarPorCodigo(ACodigo: Integer): TObject; override;
31.  function BuscarPorNome(ANome: string): TObjectList<TObject>; override;
32.  procedure EfetuarLog; override;
33.  function BuscarTodos: TObjectList<TObject>; override;
34. end;
```

Listagem 2. Classes seguindo o princípio da Segregação de Interfaces

```
1. IDAO = interface
2.   ['{8EE7C66D-7AAB-474C-8251-B7225B295A28}']
3.   procedure Salvar(AObject: TObject);
4.   procedure Excluir(AId: Integer);
5.   function BuscarPorId(AId: Integer): TObject;
6. end;
7.
8. IBuscaTodos = interface
9.   ['{1038D656-5A53-4E7A-8B06-75E9943E0E1B}']
10.  function BuscarTodos: TObjectList<TObject>;
11. end;
12.
13. IBuscaPorNome = interface
14.  ['{BDA23E97-5421-49F7-9622-44115AE7210B}']
15.  function BuscarPorNome(ANome: string): TObjectList<TObject>;
16. end;
17.
18. IBuscaPorCodigo = interface
19.  ['{44F2A9FE-7C68-47A0-A28B-3B64EBD69D91}']
20.  function BuscarPorCodigo(ACodigo: Integer): TObject;
21. end;
22.
23. ILog = interface
24.  ['{066D3F1A-72BA-4839-AC28-780832AC89BC}']
25.  function GetArquivoLog: string;
26.  procedure SetArquivoLog(const Value: string);
27.  property ArquivoLog: string read GetArquivoLog write SetArquivoLog;
28.  procedure EfetuarLog;
29. end;
30.
31. TFuncionarioDAO = class(TInterfacedObject, IDAO, ILog,
    IBuscaPorNome, IBuscaPorCodigo)
32. public
33.  procedure Salvar(AObject: TObject);
34.  procedure Excluir(AId: Integer);
35.  function BuscarPorId(AId: Integer): TObject;
36.  function BuscarPorNome(ANome: string): TObjectList<TObject>;
37.  function BuscarPorCodigo(ACodigo: Integer): TObject;
38.  procedure EfetuarLog;
39.  function GetArquivoLog: string;
40.  procedure SetArquivoLog(const Value: string);
41. end;
42.
43. TUnidadeMedidaDAO = class(TInterfacedObject, IDAO, IBuscaTodos)
44. public
45.  procedure Salvar(AObject: TObject);
46.  procedure Excluir(AId: Integer);
47.  function BuscarPorId(AId: Integer): TObject;
48.  function BuscarTodos: TObjectList<TObject>;
49. end;
```

métodos QueryInterface, _AddRef, _Release, que servem para realizar a contagem de referências das interfaces para posteriormente conseguir liberar as instâncias de forma automática quando não existem mais referências pra ela.

Além da herança de TInterfacedObject, a classe TFuncionarioDAO implementa outras interfaces que fazem sentido para ela, a IDAO para a persistência básica, ILog para fazer log de alterações na base de dados e outras duas interfaces de busca específica (IBuscaPorNome e IBuscaPorCodigo).

Desta maneira, veja que agora a classe TFuncionarioDAO implementa apenas os métodos que fazem sentido em seu escopo, o método BuscarTodos agora não é mais obrigatório e a classe consegue atender o princípio.

De maneira análoga temos a implementação de TUnidadeMedidaDAO, que agora não mais terá vários métodos que não fazem absolutamente nada, somente irá implementar os métodos da interface IDAO e implementar a interface de busca de todos os itens (IBuscaTodos), com o método BuscarTodos, atendendo também os bons princípios da orientação a objetos.

Nota

O GUID Globally Unique Identifier utilizado pelas interfaces em Delphi para garantir que exista apenas um número de referência em toda a aplicação nunca se repetindo, a probabilidade desta GUID ser gerada uma outra vez é quase impossível, tornando assim um ótimo identificador para objetos em uma aplicação.

Na **Figura 1** temos o diagrama de classes gerado a partir da codificação de nossas classes. Note que é um design muito mais organizado, onde as responsabilidades estão bem divididas.

Deixamos agora de ter uma interface muito genérica e com muitas ações, para termos mais interfaces de tamanhos menores, onde cada uma trabalha com uma única e bem definida responsabilidade, o que acaba atendendo também o princípio da responsabilidade única estudado na parte 1 do artigo.

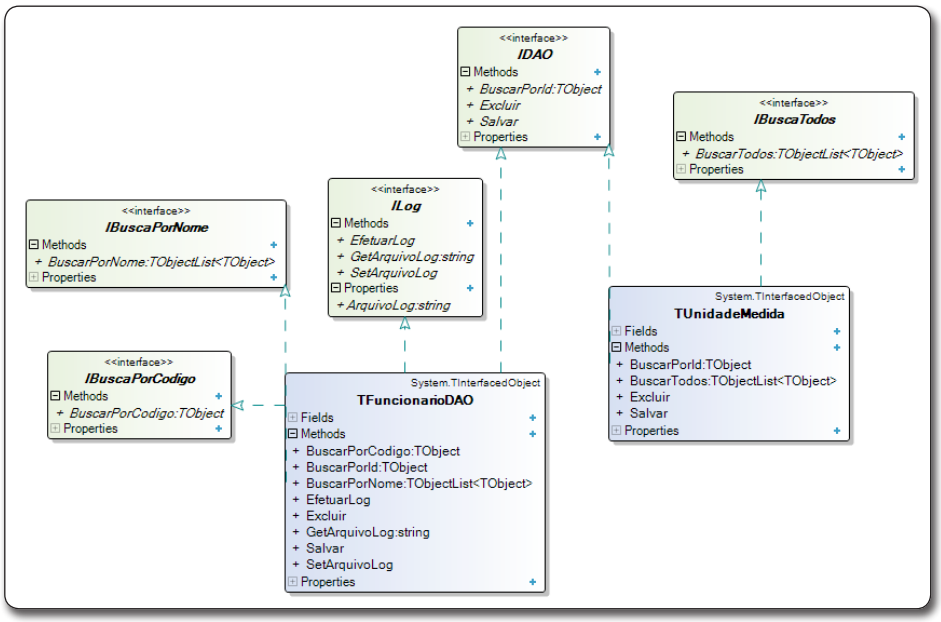


Figura 1. Diagrama de Classes da solução final

Inversão de controle e injeção de dependências na prática

Vamos exemplificar a inversão de controle e a injeção de dependências de maneira bastante simples para o leitor, e com exemplos corriqueiros do dia a dia.

Neste exemplo temos duas classes, TCliente e TEndereco, sendo que a primeira possui uma referencia para a segunda, caracterizando um relacionamento de composição entre as duas classes, pois um cliente tem um endereço, conforme pode ser visto na **Listagem 3**.

A classe TObject que é a base de todas as classes do Delphi, possui um método chamado ToString, que para representar as informações daquela classe de forma textual, por esse motivo fazemos a sobrescrita deste método (linha 10) e o implementamos da maneira desejada.

Na implementação do método ToString (linhas 35 a 47) fazemos uso da classe TStringBuilder, presente nas últimas versões do Delphi para fazermos a manipulação de strings, visto que os tipos string são imutáveis e quando fazemos uma concatenação de strings o compilador precisa realocar todo os espaços em memória daquela variável. A classe TStringBuilder utiliza um mecanismo mais eficiente na manipulação de strings e é muito recomendável nesses casos. Seu uso é simples, utilizamos o método Append para adicionarmos informações e o método ToString para pegar o resultado final das concatenações. Observe também que utilizamos o método Exit para devolver o resultado da função, este é um recurso também existente somente nas últimas versões do Delphi.

Nota

Você pode ter observado linha 1 da **Listagem 3** que temos uma pré-declaração da classe TEndereco, isso se faz necessário pela razão do compilador presente no Delphi ser top-down, ou seja de cima para baixo. Desta maneira para declararmos referências para a classe TEndereco, como nas linhas 6 e 11, precisamos informar ao compilador que esta classe será implementada abaixo. O segundo detalhe a ser notado é o uso da diretiva strict private para os atributos das classes, isso é feito para evitar o problema existente das "classes amigas", onde classes presentes numa mesma unidade tem acesso aos atributos privados de todas as classes, quebrando o encapsulamento.

Veja que na linha 12 temos uma propriedade do tipo Endereco. Neste modelo, o primeiro item a ser observado é que foi utilizada uma classe concreta para apontar qual o tipo da propriedade. Este mecanismo caracteriza um alto acoplamento entre as partes, pois estamos criando uma interdependência entre as classes por estarmos criando esta referência dentro da classe TCliente.

Outro problema pode ser visto no método construtor e destrutor, nas linhas 27 e 32,

Listagem 3. Classes TCliente e TEndereco

```
1. TEndereco = class;
2.
3. TCliente = class
4. strict private
5.   FNome: string;
6.   FEndereco: TEndereco;
7. public
8.   constructor Create;
9.   destructor Destroy; override;
10.  function ToString: string; override;
11.  property Nome: string read FNome write FNome;
12.  property Endereco: TEndereco read FEndereco write FEndereco;
13. end;
14.
15. TEndereco = class
16. strict private
17.  FLogradouro: string;
18.  FNumero: Integer;
19. public
20.  property Logradouro: string read FLogradouro write FLogradouro;
21.  property Numero: Integer read FNumero write FNumero;
22. end;
23.
24. constructor TCliente.Create;
25. begin
26.  FEndereco := TEndereco.Create;
27. end;
28.
29. destructor TCliente.Destroy;
30. begin
31.  FEndereco.Free;
32.  inherited;
33. end;
34.
35. function TCliente.ToString: string;
36. var
37.  StrBuilder: TStringBuilder;
38. begin
39.  StrBuilder := TStringBuilder.Create();
40.  try
41.   StrBuilder.Append('Cliente: ').Append(FNome).Append(sLineBreak);
42.   StrBuilder.Append('Endereco: ').Append(FEndereco.Logradouro).Append
      (', ').Append(FEndereco.Numero);
43.   Exit(StrBuilder.ToString);
44.  finally
45.   StrBuilder.Free;
46.  end;
47. end;
```

onde vemos que quem está gerenciando o ciclo de vida do objeto Endereco é a própria classe cliente. Desta forma se por alguma razão tivermos que mudar a implementação da classe Endereço, a classe Cliente também deverá ser adaptada e recompilada, algo indesejado em um bom projeto orientado a objetos.

Na **Listagem 4** temos uma classe e método de teste para nosso modelo, observe que apenas um objeto de TCliente, isto porque a classe TCliente internamente cria e destrói a classe TEndereco, desta forma podemos acessar as propriedades de endereço do cliente. No final mandamos uma mensagem para o console através do método ToString que foi sobrescrito na classe TCliente.

Injeção de Dependências via construtor

A primeira forma de injeção de dependência que será mostrada é a via construtor da classe, onde a classe TCliente irá receber uma instância da classe TEndereco no construtor, conforme **Listagem 5**.

Nessa implementação já temos a evolução do nosso modelo, de maneira que agora a classe TCliente não precisa mais cuidar do ciclo de vida da classe TEndereco, já que ela recebe uma instância da classe TEndereco via método construtor. Veja que na linha 15, precisamos criar uma instância de TEndereco antes de chamarmos o construtor da classe TCliente (linha 17) e passarmos a referência. É neste método também que cuidamos da criação e destrução do objeto TEndereco, seu ciclo de vida agora está fora da classe TCliente.

Esta já é uma evolução, já que conseguimos fazer a injeção de dependências, porém a inversão de controle ainda não foi atingida, pois existe uma dependência de TCliente para TEndereco, conforme a linha 4. Observe que estamos apontando este atributo

Listagem 4. Teste da classe TCliente do método normal

```
1. class procedure TTeste1.Teste;
2. var
3.  Cliente: TCliente;
4. begin
5.  Cliente := TCliente.Create;
6.  try
7.   Cliente.Nome := 'Maria da Silva';
8.   Cliente.Endereco.Logradouro := 'Rua Tiradentes';
9.   Cliente.Endereco.Numero := 60;
10.  Writeln(Cliente.ToString());
11.  finally
12.   Cliente.Free;
13.  end;
14. end;
```

Listagem 5. Injeção de Dependência via método Construtor

```
1. TCliente = class
2. private
3. ...
4.  FEndereco: TEndereco;
5. public
6.  constructor Create(AEndereco: TEndereco);
7. ...
8. end;
9.
10. class procedure TTeste2.Teste;
11. var
12.  Cliente: TCliente;
13.  Endereco: TEndereco;
14. begin
15.  Endereco := TEndereco.Create;
16.  try
17.   Cliente := TCliente.Create(Endereco);
18.   try
19.    ...
20.   finally
21.    Cliente.Free;
22.   end;
23.  finally
24.   Endereco.Free;
25.  end;
26. end;
```

para uma classe concreta, não para uma abstração, caracterizando assim um alto acoplamento entre elas.

Para resolvermos este problema precisamos evoluir um pouco o modelo, abstraindo a interface de TEndereco para uma interface chamada IEndereco, conforme **Listagem 6**.

Listagem 6. Novo design de classes após a refatoração

```
1. IEndereco = interface
2.   ['{5DD17698-E2B0-4488-AF2B-02695F7B632C}']
3.   function GetLogradouro: string;
4.   procedure SetLogradouro(const Value: string);
5.   function GetNumero: Integer;
6.   procedure SetNumero(const Value: Integer);
7.   property Logradouro: string read GetLogradouro write SetLogradouro;
8.   property Numero: Integer read GetNumero write SetNumero;
9. end;
10.
11. TEndereco = class(TInterfacedObject, IEndereco)
12. strict private
13.   FLogradouro: string;
14.   FNumero: Integer;
15. protected
16.   function GetLogradouro: string;
17.   procedure SetLogradouro(const Value: string);
18.   function GetNumero: Integer;
19.   procedure SetNumero(const Value: Integer);
20. end;
21.
22. TCliente = class
23. strict private
24.   FNome: string;
25.   FEndereco: IEndereco;
26. public
27.   constructor Create(AEndereco: IEndereco);
28.   function ToString: string; override;
29.   property Nome: string read FNome write FNome;
30. end;
```

Ao criarmos uma interface para a classe de endereço, criamos duas propriedades na interface IEndereco (linhas 7 e 8) e os métodos getters e setters (linhas 3 a 6) que serão responsáveis pela leitura e escritas nos atributos implementados pela classe TEndereco posteriormente.

A partir da linha 11 temos a implementação da classe TEndereco, onde ela implementa a interface IEndereco, atribuindo o valor recebido para os atributos nos métodos setters e devolvendo o valor do atributo nos métodos getters. Estas implementações ficaram fora do artigo para não o tornar muito extenso, mas as mesmas podem ser visualizadas ao efetuar o download do código fonte dos exemplos da revista na DevMedia.

A transformação da classe TEndereco para a interface IEndereco traz outro benefício que é a de o desenvolvedor não precisar mais se preocupar com a liberação da instância criada para esta interface, pois o delphi libera os recursos alocados de forma automática.

Veja como fica o uso destas últimas interfaces e classes na **Listagem 7**. Note que agora não foi mais necessário codificarmos de forma explícita a liberação da instância de endereço da memória, visto que o Delphi faz isso pra nós.

Injeção de dependências via propriedade

Esta é uma das formas mais simples de injeção de dependência e uma das preferidas pela maioria dos frameworks de injeção de dependências. Basta que adicionemos uma propriedade do tipo da interface desejada e popularmos a propriedade quando desejarmos, conforme **Listagem 8**.

Listagem 7. Uso do modelo com injeção de dependência com construtor e interface

```
1. class procedure TTeste3.Teste;
2. var
3.   Cliente: TCliente;
4.   Endereco: IEndereco;
5. begin
6.   Endereco := TEndereco.Create;
7.   Cliente := TCliente.Create(Endereco);
8.   try
9.     ...
10.  finally
11.    Cliente.Free;
12.  end;
13. end;
```

Listagem 8. Injeção de Dependências via Propriedade

```
1. TCliente = class
2. strict private
3.   FNome: string;
4.   FEndereco: IEndereco;
5. public
6.   property Nome: string read FNome write FNome;
7.   property Endereco: IEndereco read FEndereco write FEndereco;
8. end;
9.
10. class procedure TTeste4.Teste;
11. var
12.   Cliente: TCliente;
13.   Endereco: IEndereco;
14. begin
15.   Endereco := TEndereco.Create;
16.   Cliente := TCliente.Create;
17.   try
18.     Cliente.Endereco := Endereco;
19.     ...
20.   finally
21.     Cliente.Free;
22.   end;
23. end;
```

O restante da codificação fica semelhante ao da injeção de dependências via método construtor visto anteriormente. Quem desejar mais detalhes pode visualizar no código fonte junto do artigo no site da DevMedia.

No método de teste, precisamos novamente da criação da instância de Endereço, só que desta vez atribuímos a referência através da propriedade Endereco (linha 18).

Injeção de dependências via interface

A codificação da injeção de dependências através da criação de uma interface é um pouco diferente, e exige a criação de mais uma interface, conforme pode ser visto na **Listagem 9**.

Declaramos uma interface IEnderecoDI que irá servir basicamente para ser implementada quando uma classe necessita de um objeto Endereço no seu interior. Também se faz necessária a criação de um método SetEndereco que irá receber essa interface por parâmetro e armazenar na variável de referência.

Listagem 9. Injeção de Dependências via Interface

```
1. IEnderecoDI = interface
2.   ['{D545EEAC-15CA-4055-8745-B1F96BA8DBF3}']
3.   procedure SetEndereco(AEndereco: IEndereco);
4. end;
5.
6. TCliente = class(TInterfacedObject, IEnderecoDI)
7. strict private
8.   FEndereco: IEndereco;
9. public
10.   procedure SetEndereco(AEndereco: IEndereco);
11. end;
12.
13. procedure TCliente.SetEndereco(AEndereco: IEndereco);
14. begin
15.   FEndereco := AEndereco;
16. end;
17.
18. class procedure TTeste5.Teste;
19. var
20.   Cliente: TCliente;
21.   Endereco: IEndereco;
22. begin
23.   Endereco := TEndereco.Create;
24.   Cliente := TCliente.Create;
25.   try
26.     Cliente.SetEndereco(Endereco);
27.     ...
28.   finally
29.     Cliente.Free;
30.   end;
31. end;
```

Veja que agora a classe TCliente implementa a interface IEnderecoDI (linha 6) e é obrigada a implementar o método SetEndereco que recebe uma referência e guarda na variável interna FEndereco.

Agora quando desejamos fazer uso destas classes, faz-se necessário fazer chamada ao método SetEndereco da classe TCliente (linha 26) e passarmos a referência de Endereco que foi instanciada anteriormente, para que a mesma possa ser utilizada sem problemas.

Note que existem várias formas de aplicar o conceito de injeção de dependências em nossas modelagens de classe. Cabe a nós analistas de sistemas e desenvolvedores fazer uso daquele que mais admiramos ou aquele que consideremos o mais simples. Veja que a última maneira apresentada adiciona mais uma camada de abstração, mais uma interface no modelo, de certa forma aumentando um pouco a complexidade do modelo.

Este tipo de análise que deve ser feita principalmente por Analistas Desenvolvedores de Sistemas que já possuem formação e experiência no desenvolvimento de softwares, pois estão mais capacitados a fazer a identificação dos padrões de projeto e princípios da orientação a objetos que se aplicam a determinada situação.

Cada princípio SOLID tem várias formas de análise e implementação, como os domínios dos problemas que temos no dia a dia são bastante variados, indo de softwares simples como de controles de estoque até sistemas mais complexos e extensos como uma folha de pagamento ou um sistema bancário, devemos ter cuidado ao aplicar corretamente os princípios, de maneira que eles venham a ajudar e não apenas adicionar mais complexidade aos softwares.

Como pudemos observar no decorrer do artigo, uma mudança de requisito não deve ser encarada como um problema e sim uma oportunidade de melhora do software. Vimos que através de metodologias ágeis podemos atingir este objetivo.

Também fica bastante claro o papel fundamental dos princípios SOLID no bom design de um software, fazendo com que o mesmo seja mais escalável, de fácil manutenção e de código com fácil entendimento.

Ao aliarmos as metodologias ágeis aos princípios SOLID e os padrões de projeto, termos a certeza de um software eficiente. É claro que estes princípios e padrões precisam ser estudados e entendidos de forma clara, de maneira que no dia a dia o desenvolvedor saiba como e onde aplicá-lo.

Notamos também que todos os requisitos estão relacionados e no momento que ferimos um deles, acabamos ferindo outros. Portanto é necessário que sigamos a todos os princípios, para que venhamos obter sucesso no desenvolvimento de nossos aplicativos e que o cliente fique satisfeito com o resultado final, pois na prática é isto que mais importa, a satisfação de nossos clientes.

Autor

FILIPE DALEPIANE
filipe.dalepiane@gmail.com
http://filipedalepiane.blogspot.com.br/
Analista de Sistemas numa Universidade.
Colunista das revistas Clube Delphi e .NET Magazine.
Desenvolvedor Delphi e C#, utilizando bancos de dados Firebird e SQL-Server. Bacharel em Ciência da Computação. Certificação Delphi Developer

Você gostou deste artigo?

Dê seu voto em ***www.devmedia.com.br/clubedelphi/feedback***

Ajude-nos a manter a qualidade da revista!

Conhecimento faz diferença!

Faça já sua assinatura digital ! | www.devmedia.com.br/es

Faça um *upgrade* em sua carreira

Em um mercado cada vez mais focado em qualidade, ter conhecimentos aprofundados sobre requisitos, metodologia, análises, testes, entre outros, pode ser a diferença entre conquistar ou não uma boa posição profissional.
Sabendo disso a DevMedia lançou uma publicação totalmente especializada em Engenharia de Software.
Todos os meses você pode encontrar artigos sobre Metodologias Ágeis; Metodologias tradicionais (document driven); ALM (application lifecycle); SOA (aplicações orientadas a serviçoes); Análise de sistemas; Modelagem; Métricas; Orientação à Objetos; UML; testes e muito mais. **Assine Já!**

DEVMEDIA

Google Maps: criando mapas com Delphi para Android

Trace rotas, calcule distâncias, entenda a API

Um dos serviços de localização mais utilizadas no mundo, se não o mais utilizado, é o Google Maps, serviço de mapas da empresa multinacional Google. Transformando essa informação em números, percebemos mais facilmente o alcance da ferramenta. No segundo semestre de 2013, 54% (cinquenta e quatro por cento) dos usuários de smartphones utilizaram o mapa online da empresa estadunidense, segundo a consultora GlobalWebIndex. Se considerarmos a magnitude do serviço, este número é perfeitamente compreensível e de certa forma chega a ser pequeno.

Esse aplicativo é capaz de localizar qualquer lugar no mundo, exibindo imagens reais das ruas, dando ao usuário a impressão de estar no local. Além dos mapas, o aplicativo é capaz traçar várias rotas entre dois ou mais endereços, calcular a distância destes, monitorar o trânsito e até fazer um tour pelos principais pontos turísticos do mundo, exibindo imagens em 3D até mesmo do interior de alguns deles. Para facilitar mais ainda, é possível acionar o aplicativo através de comandos de voz.

Disponível na Web, Android e iOS, com essa ferramenta será possível ter o mundo inteiro ao nosso alcance.

GPS (Global Positioning System)

O Sistema de Posicionamento Global, popularmente conhecido como GPS, faz parte do cotidiano de muita gente, as informações enviadas por satélites em torno do planeta chegam às telas de milhões de dispositivos, sendo capaz de fornecer a um aparelho receptor sua posição com muita precisão, em qualquer lugar do mundo. Isso é possível devido a uma rede de satélites que envia constantemente sinais para a Terra. Cada um dos satélites está posicionado em um ângulo diferente, para que seja possível atingir os aparelhos que estão espalhados por todo o planeta. Estes aparelhos combinam os sinais dos satélites para determinar sua posição, feito isso, essa posição é representada em um mapa. A rede GPS, que é americana, é composta por 24 satélites e foi criada em 1973, possuindo uso exclusivamente militar e precisão originalmente milimétrica. Em 1980 o governo americano abriu o uso da rede GPS para fins civis e por medida de segurança, foi imposto o que eles chamam de disponibilidade seletiva, que é um erro induzido ao sinal do satélite. Assim, os satélites enviam sinais com erros variáveis, impedindo (propositalmente) que aparelhos comerciais tenham uma precisão inferior a 100 metros. Atualmente algumas empresas já conseguiram autorização junto aos americanos e corrigiram esse "erro" em seus aparelhos comerciais. Mas apesar disso, alguns aparelhos ainda erram sua posição, e os dois principais fatores que causam esse erro são o desenho mal feito do mapa ou o aparelho receptor ser de má qualidade. Alguns fatos interessantes sobre os satélites da rede GPS:

- O primeiro satélite foi lançado em 1978;
- A rede atingiu o número de 24 satélites em 1994;
- Cada satélite possui a durabilidade de 10 anos, exigindo substituições ao longo do tempo;
- A rede GPS está posicionada cerca de 19.312,128 quilômetros acima de nós;

Fique por Dentro

Esse artigo aborda de forma clara e objetiva algumas maneiras de utilizar o Google Maps em um aplicativo Android desenvolvido em Delphi. Será demonstrado como tirar proveito dos principais recursos de localização e mapeamento, agregando ao seu app funcionalidades de grande utilidade.

• Os satélites estão em constante movimento, e em menos de 24 horas realizam duas voltas completas na órbita da terra, viajando a uma velocidade de 11.265,408 Km/h.
• Os satélites funcionam através da energia solar, possuem baterias para fornecimento de energia em casos de eclipses e possuem pequenos jatos em sua estrutura que os mantém voando no caminho correto;
• Pesam cerca de 907 quilos e medem 5 metros, de uma ponta a outra de seus painéis dólares.

Existem outras redes de satélites para posicionamento global. Por exemplo, a Rússia possui a GLONASS, a China possui a COMPASS e a Europa possui a GALILEO.

Como esses mapas são feitos?

Além dos dados enviados pelos satélites espaciais, as imagens do GPS chegam até nós devido a uma equipe de profissionais que coleta informações e imagens de todos os locais possíveis. Essas informações são adquiridas durante um "passeio" por todas as ruas de uma cidade, e durante o percurso são coletadas informações como velocidade máxima permitida, sentido da via, quantidade faixas, pontos turísticos e etc. Os dados coletados são armazenados no mapa com uma localização específica, ou seja, são georreferenciados (**BOX 1**).

BOX 1. Georreferenciamento

Georreferenciamento trata do posicionamento de locais em um sistema de coordenadas geográficas escolhido. Quando um georreferenciamento é feito de forma adequada, uma medição de distância entre áreas é apurada, precisa, assim como a sobreposição geoespacial de dados de diferentes formas. Um exemplo dessa sobreposição é combinação de informações para geração de mapas específicos como mapas que mostram a densidade de chuvas por cidades. Caso as regiões afetadas pelas chuvas não estejam georreferenciadas corretamente, esse mapa pode exibir informações incorretas. Outro problema que pode ocorrer com falhas de georreferenciamento é a exibição de objetos 3D em mapas e problemas na escala de metragem. No caso dos objetos 3D, os mesmos poderiam aparecer em locais errados e no caso de topologia de terrenos, um terreno que pode aparecer deformado, mostrando elevações que não existem. No caso da escala, um metro medido pode ser na realidade, dis metros, por exemplo.

App de demonstração

Vamos criar um app que faz uso de alguns destes recursos. Nosso app será capaz de traçar rotas entre dois endereços, mostrar determinado local no mapa e também mostrar a posição atual do dispositivo. Existem diversas técnicas para fazer um app se "comunicar" com o Google Maps, aprenderemos aqui algumas delas.

Nossa Interface

Crie uma nova aplicação Firemonkey Mobile e salve-a com o nome MyMaps. Nosso aplicativo terá um visual simples e um pouco fora do padrão Android, mas como isso (padrões de layout) não é nosso foco, não há problemas. Para melhor visualização, escolha um Android Tablet 7.0 como modelo e mude sua orientação para horizontal. Agora altere a propriedade Name do Form1 para frmPrincipal, solte dentro deste uma ToolBar e altere suas propriedades Name para tbCabecalho, StyleLookup para toolbarstyle e TintColor para Skyblue. Coloque sobre a ToolBar um Label, e mude as propriedades Name para lblTitulo, Align para Center, StyleLookup para labelstyle e a propriedade Text para Google Maps. Em seguida largue um Panel no frmPrincipal com alinhamento Top, Name pnCabecalho e StyleLookup panelstyle. Agora, traga (ainda para o frmPrincipal) um WebBrowser e lhe dê um alinhamento (Align) Client e nome wb. Depois disso, solte dentro do pnCabecalho um GridPanelLayout e altere as propriedades Name e Align para gpl e Left respectivamente.

Esse componente (GridPanelLayout) permite definir um tipo de layout que chamamos de TableLayout e servirá de menu para o nosso app. Por padrão, ele vem com duas colunas definidas (**Figura 1**), altere a propriedade Value das duas colunas (TColumnItem) para 50, isso irá fazer com que cada coluna ocupe 50% (cinquenta por cento) do GridPanelLayout. Para completar a interface do nosso menu, solte quatro Images sobre ele e altere a propriedade Name de cada uma para: imgMaps, imgSensor, imgMyMaps e imgDistancia. Escolha quatro imagens de sua preferência. Ainda no pnCabecalho solte dois Edits e mude sua propriedade Name para edtOrigem e edtDestino e a propriedade TextPrompt para endereço de partida e endereço de destino. O resultado é visto na **Figura 2**.

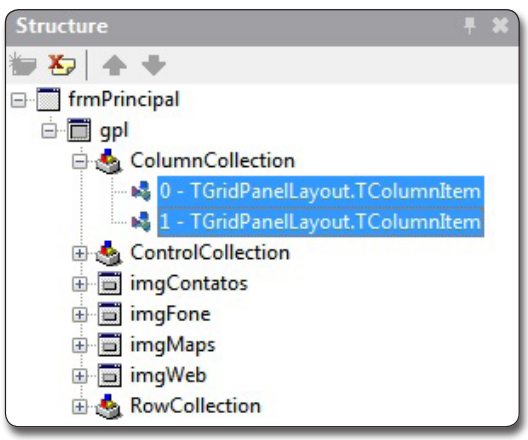


Figura 1. Colunas do GridPanelLayout

Figura 2. Interface do aplicativo

Intent

Traduzindo para o português, intent significa intenção e como o nome sugere, representa uma intenção que o aplicativo tem de realizar determinada ação. Considerada o coração do Android, a classe android.content.Intent aparece em todos os lugares, é ela a responsável por representar mensagens de uma aplicação para o sistema operacional, solicitando que algo seja realizado. Sendo assim, podemos considerar que uma Intent tem o papel de uma ponte na integração de diferentes aplicações. Resumindo, o aplicativo envia uma intenção ao sistema operacional e cabe a este interpretar esta mensagem e tomar as devidas providências, que pode ser o início de outra aplicação, abrir uma página da web ou até mesmo tirar uma foto.

A primeira funcionalidade do nosso aplicativo será a de exibir um determinado local no mapa. Faremos isso através de Intents, ou seja, vamos apenas passar nossa intenção e deixar que o Google Maps faça o resto.

Criamos um procedure na seção private do formulário chamada Maps, que possui a seguinte assinatura: **procedure** Maps(Sender: TObject); e esse procedure será executado no clique do botão imgMaps. A **Listagem 1** apresenta sua implementação.

Listagem 1. Implementação da Procedure Maps

```
1. procedure TfrmPrincipal.Maps;
2. var
3.  Intent: JIntent;
4. begin
5.  // Instanciando a classe Inten
6.  Intent := TJIntent.Create;
7.  // Passando a ação para o método setAction
8.  Intent.setAction(TJIntent.JavaClass.ACTION_VIEW);
9. // Passando a URI para o método setData
10. Intent.setData(TJnet_Uri.JavaClass
11.         .parse(StringToJString('geo://0,0?q=' +
12.          edtOrigem.Text)));
13. // Passamos nossa Inteção para o método startActivity
14. SharedActivity.startActivity(Intent);
15.end;
```

Para termos a acesso a classe Intent, declaramos uma variável do tipo JIntent (linha 3), feito isso, instanciamos nossa classe (linha 6) e logo em seguida passamos uma ação para o método setAction da Intent (linha 8). Depois disso informamos a URI para o método setData (linha 10) e por fim iniciamos nossa Intenção através do método startActivity (linha 14).

Basicamente, para enviar uma intenção para o sistema operacional, precisamos informar a ação e especificar a URI. A junção desses dois especifica nossa intenção. No nosso exemplo, passamos a ação ACTION_VIEW e a URI geo://0,0?q= mais o endereço passado no edtOrigem (poderia ser as coordenadas). Isso fará com que o aplicativo Google Maps seja iniciado e logo em seguida carregue a localização passada na URI, como pode ser visto na **Figura 3**.

Vale ressaltar que nem sempre será preciso passar uma URI, existem algumas intenções que requerem apenas a ação. Na documentação do Android (seção **Links**) é possível encontrar uma lista de Intents, demonstrando a ação e a URI que devem ser passadas. Algumas delas são:

- Câmera (Capturar Imagem ou Vídeo)
  - Ação= ACTION_IMAGE_CAPTURE ou ACTION_VIDEO_CAPTURE
  - URI= Nenhum
- Telefone (Iniciar uma Chamada telefônica)
  - Ação= ACTION_IMAGE_CAPTURE ou ACTION_VIDEO_CAPTURE
  - URI= tel://98390170
- Bateria(Status da Bateria)
  - Ação= ACTION_POWER_USAGE_SUMMARY
  - URI= Nenhum

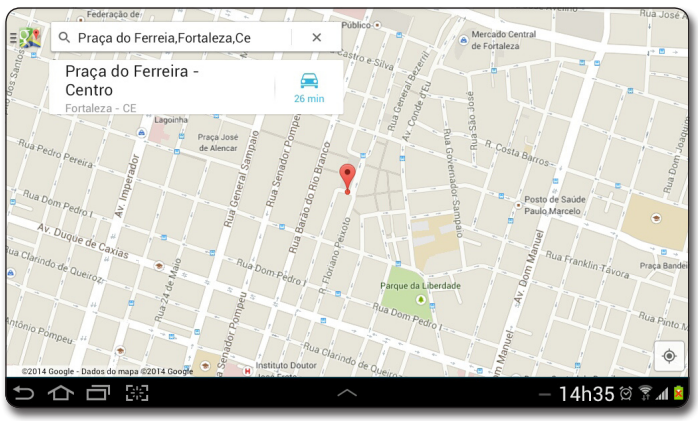


Figura 3. Google Maps exibindo o local passado como parâmetro

Como podemos ver, uma intenção permite que iniciemos uma atividade em outro aplicativo, descrevendo uma ação simples que gostaríamos de realizar (como "ver um mapa" ou "tirar uma foto"). É uma das maneiras mais simples e práticas de se usar o Google Maps.

Fazendo uso de Intents, nosso aplicativo está se comunicando com o app da Google e é capaz solicitar que este mostre qualquer localização no mapa. Vamos aumentar as funcionalidades do nosso aplicativo, para isso iremos dar a ele o "poder" de exibir a posição atual. Iremos precisar de um LocationSensor, ele irá trabalhar em dupla com o componente WebBrowser já adicionado. O primeiro faz uso do GPS do dispositivo para descobrir sua localização e o segundo exibirá esta localização. Primeiramente devemos adicionar ao formulário o componente LocationSensor e na sequência criar um procedure que o ativa. O procedure chama-se Sensor e é declarado na seção private. A **Listagem 2** exibe sua implementação.

Listagem 2. Implementação

```
procedure TfrmPrincipal.Sensor(Sender: TObject);
begin
 LocationSensor1.Active := True;
end;
```

Sua implementação é bem simples. O componente LocationSensor possui uma propriedade chamada Active, que recebe como parâmetro um valor booleano, indicando se deve ser ativado ou não. Posteriormente esse procedure é executado no evento OnClick do botão imgSensor.

Com o sensor ativado, podemos capturar a longitude e latitute atual do dispositivo através do evento OnLocationChanged do componente LocationSensor (**Listagem 3**).

Listagem 3. Evento OnLocationChanged do LocationSensor

```
1. procedure TfrmPrincipal.LocationSensor1LocationChanged(
2.   Sender: TObject;
3.   const OldLocation, NewLocation: TLocationCoord2D);
4. var
5.   url: string;
6. begin
7.   try
8.     FormatSettings.DecimalSeparator := '.';
9.     url :=
10.      Format('https://maps.google.com/maps?q=%2.6f,%2.6f',
11.             [NewLocation.Latitude,
12.              NewLocation.Longitude]);
13.    wb.Navigate(url);
14.  finally
15.    LocationSensor1.Active := False;
16.  end;
17. end;
```

Para que o Google Maps interprete de maneira correta as coordenadas passadas pelo LocationSensor, temos que alterar o nosso separador decimal, ou seja, trocar a vírgula (,) pelo ponto (.) (linha 8), em seguida passamos para a URL fornecida pela Google (linha 10), a latitude e longitude, que podemos acessar através do NewLocation. Observe que usamos um Format para que estes valores (Latitude e Longitude) tenham o formato numérico de dois inteiros e seis casa decimais. E por fim passamos a URL para o WebBrowser (linha 13), este irá exibir a posição atual do dispositivo no mapa. Optamos por desativar o LocationSensor no final do processo, mas caso fosse necessário, poderíamos deixá-lo ativo, assim, ele irá atualizar a posição sempre que houver um deslocamento de alguns metros do dispositivo. Esse deslocamento pode ser configurado usando a propriedade Distance do LocationSensor. O resultado é o que vemos na **Figura 4.**

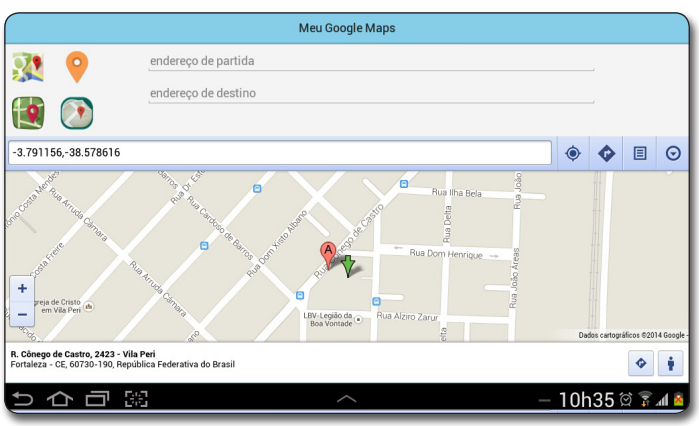


Figura 4. Posição atual do dispositivo

Iremos agora fazer uso de um dos recursos mais interessantes do Google Maps, Rotas. O processo será semelhante ao que fizemos para exibir a localização atual, partiremos de uma URL padrão e acrescentaremos o endereço de partida e o endereço de destino. Declaramos na seção private um procedure de nome Rotas, esse será executado no clique do botão imgMyMaps. A **Listagem 4** mostra sua implementação.

Declaramos duas variáveis para armazenar o endereço de origem e o endereço de destino, que serão informados pelo usuário através do edtOrigem e do edtDestino. Temos também uma variável que irá receber uma URL, que nada mais é do que o endereço do Google Maps na web, acrescido de \dir. Concatenamos com essa URL o endereço de partida e o endereço de destino e o exibimos no WebBrowser. Confira o resultado na **Figura 5**.

Listagem 4. Implementação da procedure Rotas

```
procedure TfrmPrincipal.Rotas(Sender: TObject);
var
 sOrigem, sDestino, url: string;
begin
 url := 'https://www.google.com.br/maps/dir/';
 sOrigem := edtOrigem.Text + '/';
 sDestino := edtDestino.Text + '/';
 wb.Navigate(url + sOrigem + sDestino);
end;
```

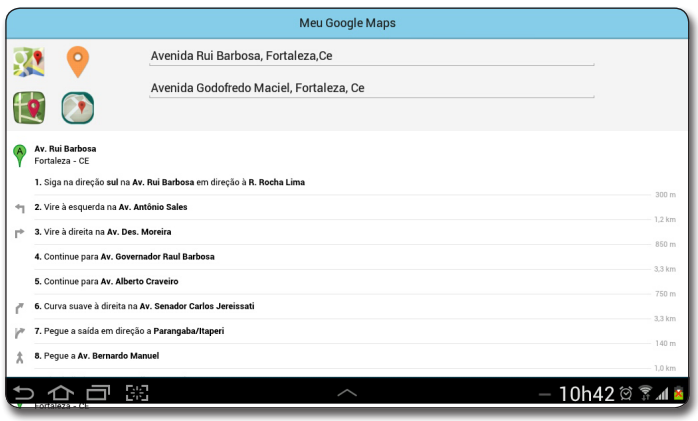


Figura 5. Aplicativo exibindo uma rota entre dois endereços

Todos os exemplos que fizemos até agora foram feitos baseados em parâmetros de URL enviados por uma solicitação HTTP. Mas o Google nos fornece uma API com a qual podemos consumir os mais variados recursos.

API (Application Programming Interface)

Uma Interface de Programação de Aplicativos é um conjunto de instruções e padrões de programação estabelecidos por um software, para que outros aplicativos possam acessar suas funcionalidades e serviços. Com as APIs, os aplicativos conversam uns com os outros sem conhecimento ou intervenção dos usuários, pois esta é composta por uma série de funções acessíveis somente por programação, e que permitem utilizar características

do software menos evidentes ao usuário comum. O usuário não percebe, mas nos bastidores há diversos aplicativos trabalhando juntos por meio de APIs.

Para facilitar o entendimento, peguemos o Windows como exemplo. Ao criarmos um programa que possua algum processo que envolva esse sistema operacional, é provável que ele faça uma conexão com alguma API do Windows. Mostrar a data e a hora é um exemplo disso.

Existem diversos serviços que disponibilizam seus códigos para serem utilizados em outros aplicativos, o Google Maps é um deles. Vamos explorar algumas APIs do Google Maps. O primeiro passo é habilitar uma API, para isso devemos ir ao Console de API (seção **Links**), onde é necessário fazer Login com uma conta Google (Gmail, por exemplo). Feito isso, procuramos pela opção Enable an API (Ativar uma API). Nessa opção encontramos uma lista de APIs com o nome, quota (quantidade de solicitações permitidas) e status (Ativa/Inativa). Provavelmente encontraremos todas as APIs com status OFF (inativa). Escolhemos a API que desejamos utilizar, para nosso exemplo escolheremos Distance Matrix API e clicamos na opção OFF, assim ela ficará ON.

Chave de API

O Google recomenda que os aplicativos que pretendam fazer uso de suas APIs, façam isso usando uma chave. Esta chave permitirá o monitoramento de nosso aplicativo, rastreando o uso da API no Console de APIs e também permite que seja adquirida cota adicional, se necessário. Através desta chave a Google pode entrar em contato conosco se for preciso.

Para gerar uma chave também usamos o Console de API. Depois de acessá-lo, localizamos no lado esquerdo o botão Credentials (Credenciais) e depois escolhemos a opção Create New Key (criar uma nova chave). Feito isso, o assistente perguntará qual tipo de chave precisamos e nos dará as seguintes opções: Server key, Browser key, Android key e iOS Key. Para nosso exemplo escolha Android Key.

Nota

Uma chave identifica a origem da solicitação e só permite que esta solicitação seja feita de um projeto que tenha o mesmo tipo da chave, ou seja, ao gerar uma Chave Android, ela só poderá ser usada em projetos da mesma plataforma, ao tentar usar em um projeto Browser, por exemplo, é gerada uma mensagem informando que a chave é inválida.

APIs do Google Maps

Com essa API fornecida pela Google, podemos criar mapas com locais definidos, utilizar controle de zoom, definir tipos de mapa, utilizar geração de rotas, pesquisar por estabelecimentos, calcular distâncias e o melhor, personalizando da maneira que acharmos melhor.

Cada API tem o seu limite de uso, ou seja, um limite de consultas, sendo que para empresas o limite é um pouco maior. Caso o usuário exceda esse limite, o servidor retornará um status HTTP 403 e também exibirá uma imagem (**Figura 6**) indicando que a cota foi ultrapassada. As APIs do Google Maps possuem uma URL padronizada, como vemos a seguir:

```
http://maps.googleapis.com/maps/api/nomeapi/output?parameters
```

Sua composição se resume em nomeapi, que é a descrição da API que deseja "consumir"; output, que é onde informamos como desejamos receber as respostas (JSON ou XML) e parameters, que são os parâmetros para busca. Cada parâmetro deve ser separado pelo símbolo do "e comercial" (&), sendo que alguns são obrigatórios.

Figura 6. Imagem retornada quando o limite de consultas é atingido

Distance Matrix API

No nosso app iremos fazer uso dessa API, então precisamos conhecer um pouco dela. Essa API nos oferece um serviço que informa a distância entre um endereço de origem e um de destino, mostrando também o tempo de deslocamento entre eles, sendo possível ainda escolher o meio de transporte e mais de um ponto de partida e de chegada.

Cada consulta enviada à API é limitada pelo número de elementos permitidos, onde o número de endereços de origens multiplicado pelo número de endereço de destinos define o número de elementos. O número máximo de elementos permitido por consulta é 100, e o limite diário de elementos é de 2.500. Há ainda uma restrição quanto ao limite de caracteres das URLs, que é de 2048 caracteres. Então, ao fazer uma consulta que envolva muitos locais, se atente a esse limite. A API de Distância tem o formato visto como a seguir:

```
http://maps.googleapis.com/maps/api/distancematrix/output?parameters
```

Como é possível ver, existe uma lista de parâmetros a serem utilizados, resumindo-se em obrigatórios e opcionais. Os obrigatórios são:

- **origins:** Um ou mais endereços separados pelo caractere de barra vertical (|), a partir dos quais a distância e o tempo são calculados. Caso queira, pode informar a latitude e longitude. Exemplo: origins= Aeroporto Pinto Martins Fortaleza;
- **destinations:** Um ou mais endereços separados pelo caractere de barra vertical (|), a partir dos quais a distância e o tempo são calculados. Caso queira, pode informa a latitude e longitude. Exemplo: origins= Arena Castelão Fortaleza;
- **sensor:** Indica se seu aplicativo está usando um sensor (por exemplo, GPS) para determinar a localização do usuário. Exemplo: sensor=false;

Os opcionais são três, listados a seguir:

• **language**: O idioma em que os resultados devem ser retornados;

• **units**: Especifica o sistema de unidades a ser usado ao expressar distância como texto;

• **mode**: Especifica qual meio de transporte usar ao calcular rotas. Os valores válidos são:

- **driving** Indica rotas de trânsito usando a rede rodoviária, este é o valor padrão;
- **walking**: Solicita rotas a pé por faixas de pedestre e calçadas (onde disponíveis);
- **bicycling**: Solicita rotas de bicicleta em ciclovias (atualmente, disponível apenas nos EUA e em algumas cidades do Canadá).

Na **Listagem 5** é possível ver um exemplo de uma solicitação completa e funcional.

Listagem 5. Solicitação

```
http://maps.googleapis.com/maps/api/distancematrix/json?origins=Aeroporto
Pinto Martins Fortaleza&destinations=Arena Castelão Fortaleza&mode=driving&
language= pt-BR&sensor=false&key=AIzxSyD-VrNG3VKRqag1Bnn1dfNYRl32qp--fl0
```

Como foi citado anteriormente, as respostas para as consultas são retornadas no formato indicado pelo sinal output, observe que passamos JSON. Estamos solicitando a distância e o tempo de viagem, partindo do aeroporto de Fortaleza (parâmetro origins) até o estádio Castelão na mesma cidade (parâmetro destinations). Solicitamos que o cálculo seja feito usando as rodovias (parâmetro mode), e que a resposta venha na língua portuguesa (parâmetro language), não estamos usando GPS para informar para informar a localização do dispositivo (parâmetro sensor) e por ultimo informamos a chave. Perceba que os parâmetros estão separados pelo caractere &.

A API do Google Maps é bastante vasta, e possui os mais variados recursos. Para usá-los basta seguir o mesmo conceito visto anteriormente, pois só o que irá mudar serão os parâmetros e nome de cada API. Para mais informações e detalhes sobre elas, visite a página (seção **Links**) de APIs do Google Maps, lá encontra-se tudo sobre essa (API de Distância) e todas as outras APIs.

Já que aprendemos alguns conceitos básicos sobre a API de Distância, vamos usá-la em nosso aplicativo. Na seção private declaramos um novo procedure, de nome Distancia. Sua implementação é vista na **Listagem 6**. Esse procedure é então chamado no evento OnClick do botão imgDistancia.

Listagem 6. Implementação da procedure Distancia

```
1. procedure TfrmPrincipal.Distancia(Sender: TObject);
2. var
3.  sOrigem, sDestino, url, sparameters: string;
4. begin
5.  url := 'http://maps.googleapis.com/maps/api/distancematrix/json?';
6.  if (edtOrigem.Text = EmptyStr) or
    (edtDestino.Text = EmptyStr) then
7.  begin
8.   ShowMessage('Informe a origem e o destino');
9.   Exit;
10. end;
11. sOrigem := 'origins=' + edtOrigem.Text + '&';
12. sDestino := 'destinations=' + edtDestino.Text + '&';
13. sparameters := 'mode=driving&language=pt-BR&sensor=false';
14. wb.Navigate(url + sOrigem + sDestino + sparameters);
15.end;
```

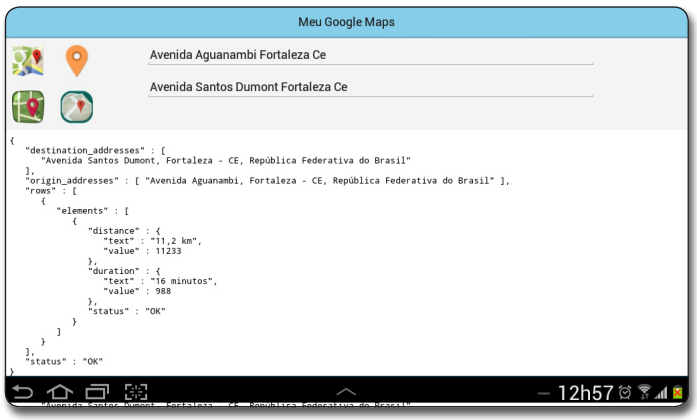


Figura 7. Resposta da API de Distancia

Atribuímos à variável url (linha 5) o formato da API, escolhendo o JSON como arquivo para resposta. Ao parâmetro origins (linha 11), que é obrigatório, passamos a propriedade Text do edtOrigem juntamente com o caractere (&) separador de parâmetros. Para o parâmetro destinations (linha 12) que também é obrigatório, fizemos a mesma coisa, só que passamos a propriedade Text do edtDestino. Passamos o meio de transporte e a linguagem para a variável sparameters (linha 13). E por fim pedimos que a resposta fosse exibida no WebBrowser (linha 14). Veja o resultado na **Figura 7.**

A resposta enviada pela API é bem fácil de ser compreendida, observe que, como passamos apenas um endereço de origem e um endereço de destino, nos foi devolvido apenas um elemento. Dentro deste elemento encontraremos o que realmente queremos, a distância e a duração de viagem entre os dois pontos. No nosso exemplo, a distância entre uma avenida e outra é de 11,2 Km e a duração do percurso é de 16 minutos.

Vimos aqui apenas um pequeno e simples exemplo de como consumir a API de Distância, mas, entendendo os conceitos passados, será possível trazer todos os recursos do Google Maps literalmente para dentro de um aplicativo, personalizando da sua maneira.

Autor

Rafael Mota
inforafael.mota@gmail.com
Desenvolvedor Delphi, estudante da linguagem Java, plataforma Android e um apaixonado por tecnologia.

Links:

Tutorial GridPanelLayout
http://blogs.embarcadero.com/sarinadupont/2014/06/16/mobile-user-interface-design-home-screen-navigation/

Console de API
https://code.google.com/apis/console

API Google Maps
https://developers.google.com/maps/?hl=pt-br

Você gostou deste artigo?

Dê seu voto em ***www.devmedia.com.br/clubedelphi/feedback***

Ajude-nos a manter a qualidade da revista!

Controles de permissão customizados no Delphi

Simplificando o processo de controle de permissões de usuário

Segurança é sempre um tema que atrai desenvolvedores, independente de plataforma ou arquitetura. Esta relação na maioria das vezes está diretamente ligada à segurança da informação, ou seja, criptografias de dados, métodos de acesso, consistência de dados e etc. Porém, o que geralmente fornecemos de segurança em relação aos nossos usuários? Quais os cuidados geralmente tomamos para que nossas aplicações forneçam recursos de restrição de acesso, ou mesmo de ações?

Ainda falando um pouco sobre segurança, podemos citar dois tipos comuns de implementação de segurança, são elas:

• **Segurança a nível de banco de dados -** Segurança a nível de banco de dados é o controle de usuários baseado nos recursos do banco. Citando o Firebird por exemplo, que é o banco de dados mais comumente adotado por desenvolvedores Delphi, temos essa implementação por meio do Security.fdb (até a versão 2.5). O Security.fdb é um banco de dados específico que armazena os usuários que têm acesso ao servidor Firebird. Seu principal problema é que, caso utilize-o como forma de controle de usuários, ele também deverá ser disponibilizado, caso contrário, será necessário fazer a criação de todos os usuários no Security.fdb (na distribuição ou nova instalação).

• **Segurança a nível de aplicação -** Segurança a nível de aplicação trata dos recursos de controle de usuário de forma convencional, na qual possuímos uma tabela de usuários no banco de dados e toda a parte de permissão e acesso é desenvolvida na própria aplicação.

Nas versões iniciais do Delphi, era bastante comum ver desenvolvedores aderindo a bibliotecas de terceiros, que forneciam diversos recursos baseados em restrição de acesso, no entanto, existem algumas desvantagens no uso de componentes de terceiros. Com o passar do tempo, quantos mais componentes registrados no IDE, mais lento é seu carregamento e manuseio, além disso, temos a dificuldade de manutenção e evolução dos componentes, entre outras dificuldades. Dentre estes componentes, destaca-se o User Control, que já foi diversas vezes abordado aqui na Clube Delphi. O User Control é um projeto Open Source que faz uso de RTTI (**BOX 1**) para realizar o mapeamento dos formulários contidos dentro de um projeto. Entre as interessantes características que este componente possui, podemos citar o mapeamento automático destes formulários, a criação de formulários de Login próprios, a criação automática das tabelas necessárias para seu funcionamento e sua compatibilidade com vários bancos de dados. Pois bem, a partir de tantas características interessantes, que tal criar seu próprio controle de usuários personalizado? É justamente a premissa de nosso projeto!

Criaremos o nosso próprio User Control, fazendo uso de conceitos da própria suíte de componentes, porém, de uma maneira simplificada e facilmente adaptável, permitindo a fácil adaptação ao perfil de sua aplicação. Sendo assim, mãos à obra!

Fique por Dentro

Segurança é um tema de extrema importância no desenvolvimento de aplicações, independente de plataforma (seja ela Desktop, Web ou Mobile). Para nós desenvolvedores, há sempre a preocupação em relação à restrição de acesso para determinados recursos, por exemplo. Imagine uma situação onde um funcionário não pode ter acesso a dados financeiros da empresa. O ideal neste caso, seria o controle de permissões que restringisse o acesso a este tipo de recurso. Esta é exatamente a proposta de nosso artigo.

BOX 1. RTTI (RunTime Type Information)

RTTI é o acrônimo para RunTime Type Information ou tipo de informação em tempo de execução em uma tradução literal. Este recurso nada mais é do que uma maneira de se obter informações relativas aos objetos e seus tipos em tempo de execução, permitindo assim manipulá-los de forma avançada.

Criando o banco de dados

Inicialmente, faremos a criação do banco de dados de nossa aplicação. Como é possível ver na **Figura 1,** teremos uma modelagem de dados simples baseada em apenas quatro tabelas. Vale salientar que as tabelas necessárias para implementação do recurso são: Formulario, Usuario e Permissao_Usuario. A tabela de Estado simplesmente está inclusa para a demonstração do mapeamento automático dos formulários, exibindo um exemplo de situação real de uma aplicação em uso.

Para o nosso banco de dados, foi adotada versão 2.5 do Firebird. O script SQL para a criação do banco encontra-se exibido na **Listagem 1**.

Construindo a aplicação

Dando início à criação de nosso exemplo, faremos uso de um conceito básico da orientação a objetos e que auxilia muito no desenvolvimento RAD (**BOX 2**): a herança visual. Sendo assim, serão criados dois modelos, um para cadastros e outro para pesquisas.

Selecione o menu File->New->VCL Forms Application – Delphi para a criação de uma nova aplicação. Salve o projeto conforme desejar, nomeie o primeiro formulário como frmPesquisa e sua unit para uPesquisa. Adicione a esta tela um componente Panel alinhado ao topo, e um DBGrid com alinhamento alClient. Dentro do Panel insira dois Label's, um Edit (edtConteudo), um ComboBox (cmbParametro) e

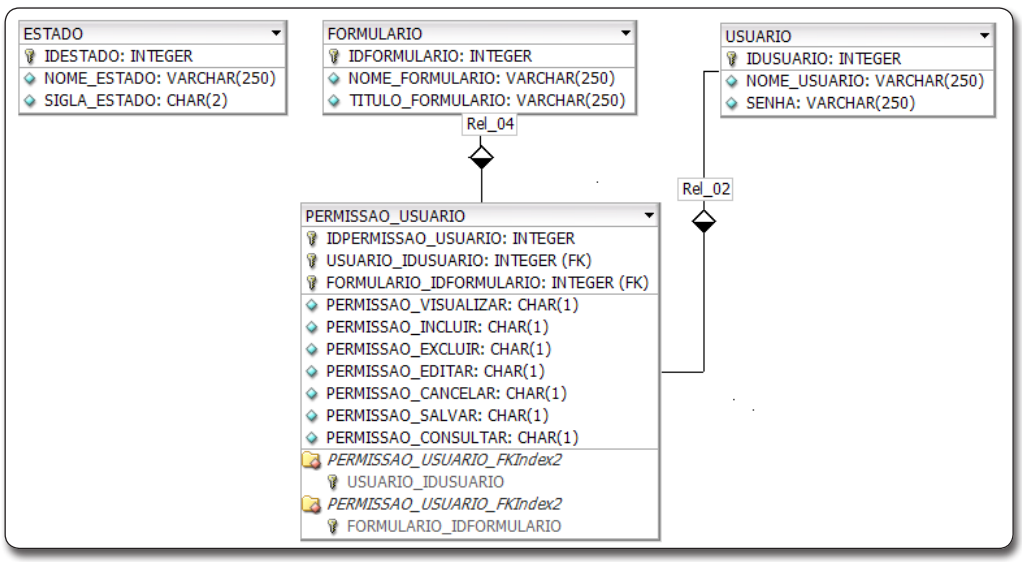


Figura 1. Modelo proposto para o exemplo

Listagem 1. Script de criação dos bancos de dados

```
SET SQL DIALECT 3;

SET NAMES WIN1252;

SET CLIENTLIB '<DIRETÓRIO DA INSTALAÇÃO DO FIREBIRD>\BIN\fbclient.dll';

CREATE DATABASE '<DIRETÓRIO DO BANCO DE DADOS>\CD161.FDB'

USER 'SYSDBA' PASSWORD 'masterkey'

PAGE_SIZE 8192

DEFAULT CHARACTER SET WIN1252;

CREATE TABLE USUARIO (
  IDUSUARIO    INTEGER NOT NULL,
  NOME_USUARIO VARCHAR(250) NOT NULL,
  SENHA        VARCHAR(250) NOT NULL
);

ALTER TABLE USUARIO ADD PRIMARY KEY (IDUSUARIO);

CREATE TABLE FORMULARIO (
  IDFORMULARIO     INTEGER NOT NULL,
  NOME_FORMULARIO  VARCHAR(250) NOT NULL,
  TITULO_FORMULARIO VARCHAR(250) NOT NULL
);

ALTER TABLE FORMULARIO ADD PRIMARY KEY (IDFORMULARIO);

CREATE TABLE ESTADO (
  IDESTADO     INTEGER NOT NULL,
  NOME_ESTADO  VARCHAR(250) NOT NULL,
  SIGLA_ESTADO CHAR(2) NOT NULL
);

ALTER TABLE ESTADO ADD PRIMARY KEY (IDESTADO);

CREATE TABLE PERMISSAO_USUARIO (
  IDPERMISSAO_USUARIO  INTEGER NOT NULL,
  IDFORMULARIO         INTEGER NOT NULL,
  IDUSUARIO            INTEGER NOT NULL,
  PERMISSAO_VISUALIZAR CHAR(1) NOT NULL,
  PERMISSAO_INCLUIR    CHAR(1) NOT NULL,
  PERMISSAO_EXCLUIR    CHAR(1) NOT NULL,
  PERMISSAO_EDITAR     CHAR(1) NOT NULL,
  PERMISSAO_CANCELAR   CHAR(1) NOT NULL,
  PERMISSAO_SALVAR     CHAR(1) NOT NULL,
  PERMISSAO_CONSULTAR  CHAR(1) NOT NULL
);

ALTER TABLE PERMISSAO_USUARIO ADD PRIMARY KEY (IDPERMISSAO_USUARIO);

ALTER TABLE PERMISSAO_USUARIO ADD CONSTRAINT FK_PERMISSAO_USUARIO_1
FOREIGN KEY (IDFORMULARIO) REFERENCES FORMULARIO (IDFORMULARIO) ON
DELETE CASCADE ON UPDATE CASCADE;

ALTER TABLE PERMISSAO_USUARIO ADD CONSTRAINT FK_PERMISSAO_USUARIO_2
FOREIGN KEY (IDUSUARIO) REFERENCES USUARIO (IDUSUARIO) ON DELETE CASCA-
DE ON UPDATE CASCADE;
```

um BitBtn (btnPesquisa), adicione por último um DataSource (dsrPesquisa). A **Figura 2** ilustra o resultado de nosso Form.

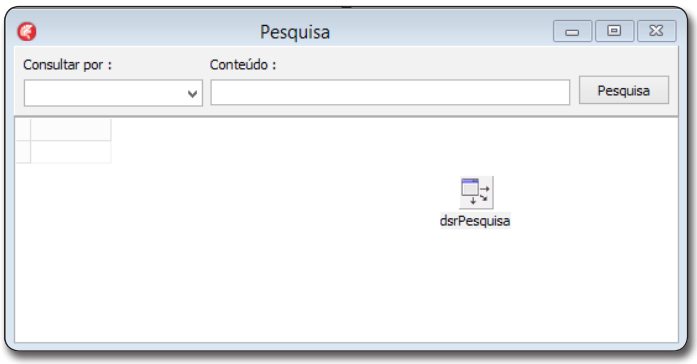


Figura 2. Formulário modelo para pesquisa

BOX 2. RAD – Rapid Application Development

Rapid Application Development ou desenvolvimento de aplicações rápidas é um recurso comum a IDE's como o Delphi e o Visual Studio, que possibilitam a criação de telas rápidas, utilizando o conceito de drag and drop (arrastar e soltar) de componentes.

Criado o primeiro modelo, passaremos para o formulário de cadastros. Crie um novo formulário acessando a opção File->New->VCL Form – Delphi. Renomeie o mesmo para frmModelo, salvando a unit com uModelo.pas. Em seguida, adicione ao formulário os seguintes componentes: um panel alinhado ao topo com seis BitBtn's (btnIncluir, btnEditar, btnExcluir, btnCancelar, btnSalvar e btnPesquisar) além de um DataSource (dsrCadastro) e um ActionList (actAcoes). Após finalizá-lo, o resultado do layout do novo formulário é ilustrado na **Figura 3**. Já na **Listagem 2** temos as implementações a serem realizadas em nosso formulário de herança.

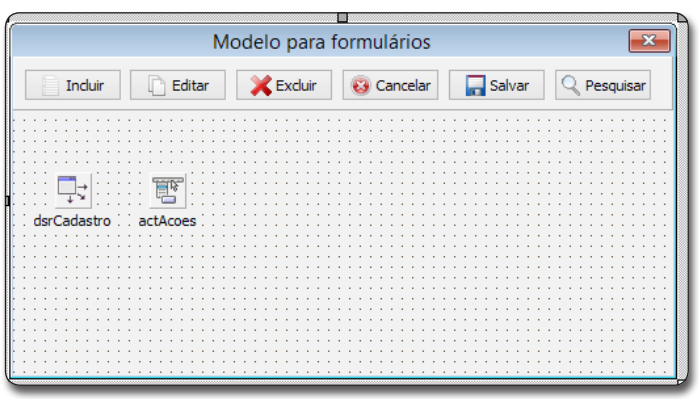


Figura 3. Formulário modelo para cadastro

Observe que no código apresentado nessa listagem temos a implementação de dois métodos (linhas 05 e 06 respectivamente) e duas propriedades (linhas 10 a 11). As linhas 02 a 04 são os getters e setters gerados a partir da propriedade. Começaremos implementando as ações mais simples do formulário, conforme a **Listagem 3**.

Listagem 2. Métodos e variáveis definidas no formulário modelo

```
...
1.  private
2.    FDataSetCadastro: TClientDataSet;
3.    FDataSetPermissao: TClientDataSet;
4.    procedure SetDataSetCadastro(const Value: TClientDataSet);
5.    procedure ApplyGrants;
6.    function IsValidDataSet(DataSet : TClientDataSet) : Boolean;
7.    procedure SetDataSetPermissao(const Value: TClientDataSet);
8.    { Private declarations }
9.  public
10.    property DataSetCadastro : TClientDataSet read FDataSetCadastro write
       SetDataSetCadastro;
11.    property DataSetPermissao : TClientDataSet read FDataSetPermissao
       write SetDataSetPermissao;
12.    { Public declarations }
13. end;
```

Listagem 3. Primeiras implementações do formulário

```
1.  function TfrmModelo.IsValidDataSet(DataSet : TClientDataSet) : Boolean;
2.  begin
3.    Result := DataSet <> nil;
4.  end;

5.  procedure TfrmModelo.SetDataSetCadastro(const Value: TClientDataSet);
6.  begin
7.    FDataSetCadastro := Value;
8.  end;

9.  procedure TfrmModelo.SetDataSetPermissao(const Value: TClientDataSet);
10. begin
11.   FDataSetPermissao := Value;
12. end;
```

Essa listagem aborda três implementações: pode-se notar entre as linhas 01 e 04 a definição da função IsValidDataSet, que como o próprio nome sugere, valida um DataSet recebido por parâmetro, verificando se o mesmo não está nulo (linha 03). Caso o DataSet não esteja nulo, o valor True é retornado pela função, afirmando que é de fato é um DataSet válido. Entre as linhas 05 a 12 tem-se apenas a atribuição dos valores recebidos por parâmetro para os atributos internos (FDataSetCadastro e FDataSetPermissao).

Finalizado, a seguir, definiremos algumas ações através de um duplo clique no componente actAcoes. A **Figura 4** ilustra as ações que devem ser definidas.

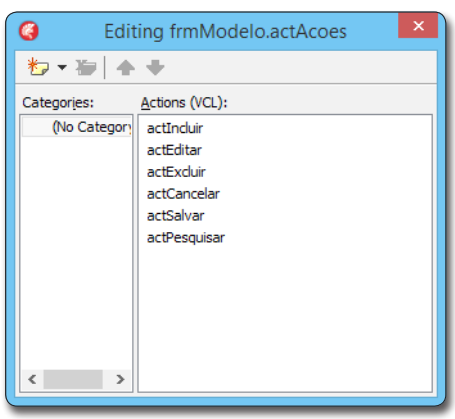


Figura 4. Ações definidas no componente actAcoes do formulário modelo

A partir da **Listagem 4** faremos as implementações destas ações uma a uma, sendo assim, dê um duplo clique sobre a actIncluir e codifique o evento. Cada ação será explanada na sequência.

Observe que na linha 03 é utilizada a função IsValidDataSet para verificar o FDataSetCadastro, evitando violações de acesso. Se o DataSet é válido, ele simplesmente é fechado, aberto e realizado um append, para que uma linha em branco seja adicionada ao mesmo (linhas 05 a 07 respectivamente).

Já na **Listagem 5** vemos a ação referente a edição dos dados. Novamente na linha 03 a função IsValidDataSet é reutilizada. Segundo uma das boas práticas de desenvolvimento de Software, devemos transformar rotinas repetitivas em métodos reutilizáveis, assim tornamos o código menor, mais simples e menos repetitivo. Na linha 05 validamos se o DataSet está ativo, para em seguida colocá-lo em modo de edição (linha 06).

Listagem 4. Evento actIncluirExecute

```
1. procedure TfrmModelo.actIncluirExecute(Sender: TObject);
2. begin
3.   if IsValidDataSet(FDataSetCadastro) then
4.   begin
5.     FDataSetCadastro.Close;
6.     FDataSetCadastro.Open;
7.     FDataSetCadastro.Append;
8.   end;
9. end;
```

Listagem 5. Código do evento actEditarExecute

```
1. procedure TfrmModelo.actEditarExecute(Sender: TObject);
2. begin
3.   if IsValidDataSet(FDataSetCadastro) then
4.   begin
5.     if FDataSetCadastro.Active then
6.       FDataSetCadastro.Edit;
7.   end;
8. end;
```

Continuando, o próximo evento abordado será o método Excluir que encontra-se na **Listagem 6**. Observe que na linha 03 ainda é utilizada a função IsValidDataSet, logo em seguida, na linha 05, é verificado se o DataSetCadastro está ativo, exibindo uma mensagem de confirmação (linha 07). Caso o usuário confirme a exclusão, o DataSet executa respectivamente os métodos Delete (linha 09) e ApplyUpdates (linha 10). Caso o número de erros seja equivalente a zero é exibida a mensagem de confirmação (linha 11), senão, é mostrada a mensagem de erro ao excluir o registro.

O método Cancelar encontra-se na **Listagem 7**. Novamente é utilizado a função IsValidDataSet, a seguir, é verificado se o DataSetCadastro encontra-se em estado de edição ou inserção (linha 05), executando os comandos Cancel e CancelUpdates que desfaz todas as alterações pendentes armazenadas no cache (linhas 07 e 08).

Para finalizar os códigos do ActionList, vamos implementar o método Salvar que encontra-se na **Listagem 8**.

Listagem 6. Código do evento actExcluirExecute

```
1.  procedure TfrmModelo.actExcluirExecute(Sender: TObject);
2.  begin
3.    if IsValidDataSet(FDataSetCadastro) then
4.    begin
5.      if FDataSetCadastro.Active then
6.      begin
7.        if MessageDlg('Deseja excluir o registro ?', mtConfirmation,
          mbYesNo, 0) = mrYes then
8.        begin
9.          FDataSetCadastro.Delete;
10.         if FDataSetCadastro.ApplyUpdates(0) = 0 then
11.           MessageDlg('Dados excluídos com sucesso !', mtInformation,
              [mbOk], 0)
12.         else
13.           MessageDlg('Problemas ao excluir o registro !', mtError,
              [mbOK], 0);
14.       end;
15.     end;
16.   end;
17. end;
```

Listagem 7. Código do evento actCancelarExecute

```
1.  procedure TfrmModelo.actCancelarExecute(Sender: TObject);
2.  begin
3.    if IsValidDataSet(FDataSetCadastro) then
4.    begin
5.      if FDataSetCadastro.State in [dsEdit, dsInsert] then
6.      begin
7.        FDataSetCadastro.Cancel;
8.        FDataSetCadastro.CancelUpdates;
9.      end;
10.   end;
11. end;
```

Listagem 8. Código do evento actSalvarExecute

```
1.  procedure TfrmModelo.actSalvarExecute(Sender: TObject);
2.  begin
3.    if IsValidDataSet(FDataSetCadastro) then
4.    begin
5.      if FDataSetCadastro.State in [dsEdit, dsInsert] then
6.      begin
7.        if MessageDlg('Deseja salvar os dados?', mtConfirmation, [mbYes,
          mbNo] ,0) = mrYes then
8.        begin
9.          FDataSetCadastro.Post;
10.         if FDataSetCadastro.ApplyUpdates(0) = 0 then
11.           MessageDlg('Dados salvos com sucesso!',
              mtInformation, [mbOK], 0)
12.         else
13.           MessageDlg('Problema ao salvar os dados!', mtError, [mbOk], 0);
14.       end;
15.     end;
16.   end;
17. end;
```

Semelhante ao método Excluir, o código apresentado testa primeiramente o DataSet através de nossa função (linha 03), em seguida, verifica se o DataSet encontra-se em estado de edição ou inserção (linha 05), exibindo uma mensagem de confirmação (linha 07), no entanto, desta vez é realizado o comando Post e em seguida o ApplyUpdates (linhas 09 e 10), exibindo uma mensagem de sucesso ou erro caso contrário.

Para finalizarmos a codificação da tela de herança, o último método a ser implementado é o procedure ApplyGrants, que tem a responsabilidade de habilitar/desabilitar os botões conforme as permissões do usuário logado no sistema (**Listagem 9**).

Abordando o procedure ApplyGrants, temos um código simples. Após verificar se o DataSet é válido e se encontra ativo, checamos se é encontrada a classe de acordo com o formulário que está válido (linha 07). Isso será visto mais adiante no decorrer do artigo. Observe que entre as linhas 09 a 14 as ações são habilitadas ou desabilitadas conforme os valores dos respectivos campos do DataSet. Finalizando, temos apenas a chamada deste método no evento OnShow do formulário, conforme mostrado na **Listagem 10**.

Após concluir as codificações, basta atribuir cada ação do ActionList ao botão correspondente.

Listagem 9. Procedure ApplyGrants

```
1. procedure TfrmModelo.ApplyGrants;
2. begin
3.   if IsValidDataSet(FDataSetPermissao) then
4.   begin
5.     if FDataSetPermissao.Active  then
6.     begin
7.       if FDataSetPermissao.Locate('NOME_FORMULARIO',
         Self.ClassName, []) then
8.       begin
9.         actIncluir.Enabled   := FDataSetPermissao.
         FieldByName('PERMISSAO_INCLUIR').AsString = 'T';
10.        actEditar.Enabled    := FDataSetPermissao.
         FieldByName('PERMISSAO_EDITAR').AsString = 'T';
11.        actExcluir.Enabled   := FDataSetPermissao.
         FieldByName('PERMISSAO_EXCLUIR').AsString = 'T';
12.        actCancelar.Enabled  := FDataSetPermissao.
         FieldByName('PERMISSAO_CANCELAR').AsString = 'T';
13.        actSalvar.Enabled    := FDataSetPermissao.
         FieldByName('PERMISSAO_SALVAR').AsString = 'T';
14.        actPesquisar.Enabled := FDataSetPermissao.
         FieldByName('PERMISSAO_CONSULTAR').AsString = 'T';
15.      end;
16.    end;
17.  end;
18. end;
```

Listagem 10. Evento onShow do formulário de herança

```
1. procedure TfrmModelo.FormShow(Sender: TObject);
2. begin
3.   ApplyGrants;
4. end;
```

Utilizando o RTTI para obter as informações de classes registradas

Como já mencionado no início, o componente User Control faz uso de um recurso para obter os formulários de maneira automática. Através de uma rápida busca pela internet, é possível ver inúmeros códigos que listam as classes registradas. Quando nos referimos ao termo "classes registradas", fazemos menção ao método RegisterClass(Classe). Normalmente, vemos esta rotina na sessão initialization dos formulários, algo semelhante a:

```
initialization
 RegisterClass(TForm);
```

Sendo assim, utilizaremos uma Unit que receberá como parâmetro uma StringList, devolvendo-a preenchida com todas as classes registradas no projeto. Crie em seu projeto uma nova unit (File->New->Unit – Delphi). Seu código é exibido na **Listagem 11** e suas implementações são abordadas mais especificamente na **Listagem 12**.

Listagem 11. Declaração da unit uEnumClasses para listar as classes registradas via RTTI

```
1. unit uEnumClasses;
2. interface
3. uses Classes;
4. procedure EnumClasses(Strings: TStrings);
5. implementation
6. type
7.   TClassesEnum = class(TObject)
8.   private
9.     FStrings : TStrings;
10.    procedure GetClassesProc(AClass: TPersistentClass);
11.  public
12.    procedure EnumClasses(Strings: TStrings);
13.  end;
```

Listagem 12. Implementações da unit uEnumClasses

```
1.  procedure EnumClasses(Strings: TStrings);
2.  begin
3.    with TClassesEnum.Create do
4.    try
5.      EnumClasses(Strings);
6.    finally
7.      Free;
8.    end;
9.  end;
10. procedure TClassesEnum.GetClassesProc(AClass: TPersistentClass);
11. begin
12.   FStrings.Add(AClass.ClassName);
13. end;
14. procedure TClassesEnum.EnumClasses(Strings: TStrings);
15. begin
16.   if not Assigned(Strings) then
17.     Exit;
18.   FStrings := Strings;
19.   with TClassFinder.Create(nil, True) do
20.   try
21.     GetClasses(GetClassesProc);
22.   finally
23.     Free;
24.   end;
25. end;
```

No código temos as implementações da classe TClassesEnum. Entre as linhas 01 e 09 temos o método construtor da classe, que a cria e executa o procedure EnumClasses, passando a StringList recebida na chamada como parâmetro. Entre as linhas 10 e 13 temos o método GetClassesProc, que adiciona as classes encontradas à instância interna de FStrings (linha 12), e por último, entre as linhas 14 e 25 tem-se a implementação de EnumClasses. Primeiramente é verificado se a StringList é válida, executando o comando Exit para sair fora do processo em caso contrário (linhas 16 e 17). A seguir, a instância interna recebe a StringList vinda por parâmetro (linha 18) e, por último, é utilizada a classe TClassFinder que executa o método GetClasses passando o procedure GetClassesProc como parâmetro.

Com isso, temos finalizado o principal recurso do artigo, o próximo passo é definir sua utilização em nosso projeto. Passaremos então para a criação da nossa camada de dados, através do Data Module.

Criando o Data Module

Finalizada a unit que será utilizada para a leitura dos formulários, iremos dar início à camada de dados. Para isso, crie um novo Data Module em File->New->Other->Delphi Projects->Delphi Files->Data Module. Renomeie-o para DMPrincipal e sua unit como uDMPrincipal.pas.

Feito isso, faremos a conexão com o banco de dados, utilizando o FireDAC. Adicione primeiramente ao DM um TFDConnection, um TFDPhysFBDriverLink e um TFDGUIxWaitCursor. A **Figura 3** ilustra os três componentes adicionados. Como o processo de configuração de conexão é muito simples e padrão para praticamente todas as engines de acesso a dados, não abordaremos a fundo, passando diretamente para a adição dos demais componentes que farão acesso direto aos dados.

Figura 3. Componentes iniciais para realizar a conexão

Após inserir e configurar a conexão, faremos a adição dos demais componentes. Insira primeiramente sete TFDQuery apontando para a conexão principal. A seguir, adicione cinco TDataSetProvider, seis TClientDataSet e um TDataSource. Para os DataSetProviders, basta apontar a propriedade DataSet para suas respectivas Queries e os ClientDataSets com seu ProviderName apontando para os DataSetProviders. A **Figura 4** demonstra a disposição e os nomes que foram utilizados para os componentes, configure-os conforme mostrado. A única particularidade se dará no cadastro de usuário, onde teremos um Mestre-Detalhe, sendo assim, configure o trio de usuário normalmente, já no DataSource, informe sua propriedade DataSet apontando para o qryCadUsuario e o qryCadUsuarioPermissao.MasterSource apontando para o dsrCadUsuario.

Listagem 13. Instruções SQL de acordo com os respectivos componentes

```
[qryCadEstado]
SELECT
  ESTADO.IDESTADO,
  ESTADO.NOME_ESTADO,
  ESTADO.SIGLA_ESTADO
FROM
  ESTADO
WHERE
  ESTADO.IDESTADO = :IDESTADO

[qryCadUsuario]
SELECT
  USUARIO.IDUSUARIO,
  USUARIO.NOME_USUARIO,
  USUARIO.SENHA
FROM
  USUARIO
WHERE
  USUARIO.IDUSUARIO = :IDUSUARIO

[qryCadUsuarioPermissao]
SELECT
  PERMISSAO_USUARIO.IDPERMISSAO_USUARIO,
  PERMISSAO_USUARIO.IDFORMULARIO,
  PERMISSAO_USUARIO.IDUSUARIO,
  PERMISSAO_USUARIO.PERMISSAO_CONSULTAR,
  PERMISSAO_USUARIO.PERMISSAO_INCLUIR,
  PERMISSAO_USUARIO.PERMISSAO_EXCLUIR,
  PERMISSAO_USUARIO.PERMISSAO_CANCELAR,
  PERMISSAO_USUARIO.PERMISSAO_SALVAR,
  PERMISSAO_USUARIO.PERMISSAO_EDITAR,
  PERMISSAO_USUARIO.PERMISSAO_VISUALIZAR,
  FORMULARIO.NOME_FORMULARIO,
  FORMULARIO.TITULO_FORMULARIO
FROM
  PERMISSAO_USUARIO
LEFT OUTER JOIN FORMULARIO ON
  PERMISSAO_USUARIO.IDFORMULARIO = FORMULARIO.IDFORMULARIO
WHERE
  PERMISSAO_USUARIO.IDUSUARIO = :IDUSUARIO

[qryFormularios]
SELECT
  FORMULARIO.IDFORMULARIO,
  FORMULARIO.NOME_FORMULARIO,
  FORMULARIO.TITULO_FORMULARIO
FROM
  FORMULARIO
ORDER BY 3

[qryConfiguraPermissao]
SELECT
  PERMISSAO_USUARIO.IDPERMISSAO_USUARIO,
  PERMISSAO_USUARIO.IDFORMULARIO,
  PERMISSAO_USUARIO.IDUSUARIO,
  PERMISSAO_USUARIO.PERMISSAO_CONSULTAR,
  PERMISSAO_USUARIO.PERMISSAO_INCLUIR,
  PERMISSAO_USUARIO.PERMISSAO_EXCLUIR,
  PERMISSAO_USUARIO.PERMISSAO_CANCELAR,
  PERMISSAO_USUARIO.PERMISSAO_SALVAR,
  PERMISSAO_USUARIO.PERMISSAO_EDITAR,
  PERMISSAO_USUARIO.PERMISSAO_VISUALIZAR,
  FORMULARIO.NOME_FORMULARIO,
  FORMULARIO.TITULO_FORMULARIO
FROM
  PERMISSAO_USUARIO
LEFT OUTER JOIN FORMULARIO ON
  PERMISSAO_USUARIO.IDFORMULARIO = FORMULARIO.IDFORMULARIO
WHERE
  PERMISSAO_USUARIO.IDUSUARIO = :IDUSUARIO

[qryLogin]
SELECT
  USUARIO.IDUSUARIO,
  USUARIO.NOME_USUARIO
FROM
  USUARIO
WHERE
  USUARIO.NOME_USUARIO = :NOME_USUARIO AND
  USUARIO.SENHA = :SENHA
```

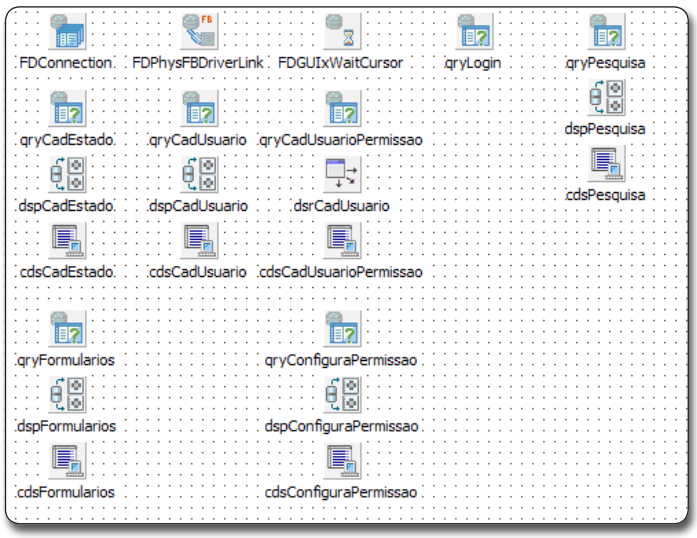


Figura 4. Componentes inseridos no DMprincipal

Após adicionados os componentes, devemos setar todas as instruções SQL de acordo com cada um dos componentes. A **Listagem 13** exibe as sentenças SQL para todos eles.

Informadas as instruções SQL em seus devidos componentes, vamos passar para a parte de implementações do DM. O Data Module concentrará grande parte das funcionalidades de nossa aplicação, sendo a parte mais trabalhosa. Na **Listagem 14** temos os atributos e métodos que devem ser declarados em seus respectivos escopos. Abordaremos a seguir cada implementação nas próximas listagens.

Podemos notar dois atributos (linhas 03 e 04) que serão utilizados para armazenar o usuário autenticado (veremos posteriormente). O método CreateGrants (linha 05), será utilizado para criar a permissão do usuário, a função GetID retornará o código do usuário autenticado (linha 06). Já na sessão public do DM, teremos o método ReadFormsAndCreateGrants, que pode ser chamado de forma externa e fará toda vez a leitura dos formulários criados, definindo as permissões de forma automática (linha 10). Na linha 11 a função CanLogin, que basicamente realiza a autenticação do usuário, permitindo ou não o acesso e, por último, os métodos SearchUserById e SearchUserByUserName que serão utilizados para busca, na pesquisa de usuários, também vista posteriormente. A função CanCreateForms (linha 17) é definida fora dos escopos, pois será uma rotina utilizada para toda a aplicação. Este método é abordado logo a seguir, na **Listagem 15**.

A função CanCreateForms terá a finalidade de restringir ou não o acesso de acordo com o formulário passado como parâmetro. Note que na linha 04 validamos se o DataSet responsável por armazenar as permissões do usuário logado está ativo, sendo afirmativa a condição, executamos uma instrução Locate na busca do formulário dentro do DataSet (linha 06). O retorno verdadeiro ou falso indicando a possibilidade de acesso é dado de acordo com o valor do campo PERMISSAO_VISUALIZAR, que deverá estar setado como 'T' para que seja possível abrir o formulário.

Listagem 14. Implementações no DataModule

```
1. ...
2.  private
3.   FIdUser: Integer;
4.   FUserName: String;
5.   procedure CreateGrants;
6.   function getID(KeyName : String) : Integer;
7.   { Private declarations }
8.  public
9.   { Public declarations }
10.   procedure ReadFormsAndCreateGrants;
11.   function CanLogin(UserName, Password : String) : Boolean;
12.   procedure SearchUserById(Id : Integer);
13.   procedure SearchUserByUserName(UserName : String);
14.  end;
15. var
16.   DMPrincipal: TDMPrincipal;
17. function CanCreateForms(ClassName : String) : Boolean;
```

Listagem 15. Função CanCreateForms

```
1.  function CanCreateForms(ClassName : String) : Boolean;
2.  begin
3.   Result := False;
4.   if DmPrincipal.cdsConfiguraPermissao.Active then
5.   begin
6.    if DMPrincipal.cdsConfiguraPermissao.Locate('NOME_FORMULARIO',
      UpperCase(ClassName), []) then
7.     Result := DMPrincipal.cdsConfiguraPermissao.
       FieldByName('PERMISSAO_VISUALIZAR').AsString = 'T'
8.    else
9.     MessageDlg('O usuário ' + DMPrincipal.FUserName +
      ' não possui acesso a este formulário!', mtWarning, [mbOk], 0);
10.  end
11. end;
```

Caso o usuário não tenha permissão, automaticamente é exibida uma mensagem de erro.

O próximo passo é a função CanLogin, descrita na **Listagem 16**. Este método como já mencionado, validará se o usuário e senha informados são validos, possibilitando ou não o acesso ao sistema. Lembrando que o formulário de Login será criado mais adiante.

A função CanLogin será utilizada posteriormente em nosso formulário de Login. Nela são recebidos dois parâmetros de entrada (usuário e senha – linha 01). Observe que entre as linhas 05 e 08 os parâmetros são passados para qryLogin, que simplesmente é aberta. O resultado é dados considerando o valor do campo query, que não deve ser nulo (linha 09). Já nas linhas 10 a 14 é testado se o Result é verdadeiro, atribuindo assim os valores dos campos usuário e senha para os atributos internos, tudo isto dentro de um bloco try finally, garantindo que a qryLogin sempre encerre sua conexão (linha 16).

Continuando a implementação, a **Listagem 17** ilustra o método getID que é responsável por retornar os valores obtidos por meio dos Generators do banco de dados.

A função getID será muito utilizada mais adiante. Ela basicamente será a responsável por retornar todos os valores das chaves primárias, sendo assim, o parâmetro necessário é a própria KeyName (linha 01). Na linha 06 é criada uma Query em runtime,

simplesmente para executar a instrução que dispara um Generator. Nas linhas 08 e 09 a conexão é atribuída e a instrução atribuída, concatenando-se a KeyName do parâmetro para a formação da SQL, por último a Query é aberta (linha 10). O Result devolverá o valor retornado pela função GEN_ID (linha 11). As linhas 13 e 14 respectivamente fecham a Query e liberam a mesma da memória.

Listagem 16. Função CanLogin

```
1. function TDMPrincipal.CanLogin(UserName, Password: String): Boolean;
2. begin
3.   Result := False;
4.   try
5.     qryLogin.Close;
6.     qryLogin.ParamByName('NOME_USUARIO').AsString := UserName;
7.     qryLogin.ParamByName('SENHA').AsString := Password;
8.     qryLogin.Open;
9.     Result := not (qryLogin.Fields[0].IsNull);
10.    if Result then
11.    begin
12.      FIdUser  := qryLogin.FieldByName('IDUSUARIO').AsInteger;
13.      FUserName := qryLogin.FieldByName('NOME_USUARIO').AsString;
14.    end;
15.  finally
16.    qryLogin.Close;
17.  end;
18. end;
```

Listagem 17. Método getID que retorna o valor da chave primária

```
1. function TDMPrincipal.getID(KeyName: String): Integer;
2. var
3.   Query : TFDQuery;
4. begin
5.   Result := -1;
6.   Query := TFDQuery.Create(nil);
7.   try
8.     Query.Connection := FDConnection;
9.     Query.SQL.Text := 'SELECT GEN_ID(GEN_' + KeyName + ',1) FROM
       RDB$DATABASE';
10.    Query.Open();
11.    Result := Query.Fields[0].Value;
12.  finally
13.    Query.Close;
14.    FreeAndNil(Query);
15.  end;
16. end;
```

O próximo passo nas implementações do Data Module é a criação do procedure CreateGrants, que será utilizada em conjunto com o procedure ReadFormsAndCreateGrants. Esta será a segunda maior implementação do nosso exemplo e terá por finalidade conceder as permissões dos formulários para os usuários. Veremos ela mais detalhadamente na **Listagem 18**.

Basicamente é realizado o cadastro automático dos formulários para cada usuário que se loga no sistema. Primeiramente o ClientDataSet cdsConfiguraPermissao é aberto, passando como parâmetro o Id do usuário autenticado (linhas 03 a 06). Em seguida, realizamos um loop no cdsFormularios, percorrendo todo o DataSet do início ao fim (linhas 08 a 10). Caso o formulário não seja localizado dentro do cdsConfiguraPermissao, ele será automaticamente cadastrado, recebendo o Id do formulário e o Id do usuário no processo (linhas 14 a 25). Repare que na linha 15 é utilizada a função GetID criada anteriormente, o outro detalhe a se atentar é que aqui definimos as permissões para todos os campos, no entanto, você pode marcar as restrições iniciais conforme quiser. No término do processo, o cdsConfiguraPemissao persiste os dados através do ApplyUpdates (linha 29) e finaliza habilitando os controles do cdsFormularios (linha 31).

Listagem 18. Método CreateGrants que define as permissões

```
1. procedure TDMPrincipal.CreateGrants;
2. begin
3.   cdsConfiguraPermissao.Close;
4.   cdsConfiguraPermissao.FetchParams;
5.   cdsConfiguraPermissao.Params[0].Value := FIdUser;
6.   cdsConfiguraPermissao.Open;
7.   try
8.     cdsFormularios.DisableControls;
9.     cdsFormularios.First;
10.    while not cdsFormularios.Eof do
11.    begin
12.      if not cdsConfiguraPermissao.Locate('IDFORMULARIO',
         cdsFormulariosIDFORMULARIO.AsInteger, []) then
13.    begin
14.      cdsConfiguraPermissao.Append;
15.      cdsConfiguraPermissaoIDPERMISSAO_USUARIO.Value :=
         getID('IDPERMISSAO_USUARIO');
16.      cdsConfiguraPermissaoIDUSUARIO.Value := FIdUser;
17.      cdsConfiguraPermissaoIDFORMULARIO.Value :=
         cdsFormulariosIDFORMULARIO.Value;
18.      cdsConfiguraPermissaoPERMISSAO_CONSULTAR.AsString := 'T';
19.      cdsConfiguraPermissaoPERMISSAO_INCLUIR.AsString := 'T';
20.      cdsConfiguraPermissaoPERMISSAO_EXCLUIR.AsString := 'T';
21.      cdsConfiguraPermissaoPERMISSAO_CANCELAR.AsString := 'T';
22.      cdsConfiguraPermissaoPERMISSAO_SALVAR.AsString := 'T';
23.      cdsConfiguraPermissaoPERMISSAO_EDITAR.AsString := 'T';
24.      cdsConfiguraPermissaoPERMISSAO_VISUALIZAR.AsString := 'T';
25.      cdsConfiguraPermissao.Post;
26.    end;
27.    cdsFormularios.Next;
28.  end;
29.  cdsConfiguraPermissao.ApplyUpdates(0);
30.  finally
31.    cdsFormularios.EnableControls;
32.  end;
33. end;
```

Aproveitando a deixa, agora criaremos o método mais importante de nossa aplicação e que faz uso do procedure da listagem anterior, ReadFormsAndCreateGrants, e faz a leitura dos formulários e de seus Captions de forma dinâmica. Veremos como é feito o processo na **Listagem 19**.

Entre as linhas 02 a 07 a definição das variáveis. Na linha 09 está a StringList que receberá as classes registradas no sistema. A seguir, o cdsFormularios é fechado e aberto novamente para que os formulários sejam atualizados (linha 12 e 13). Na linha 14 a classe EnumClasses é chamada através do método de mesmo nome, passando o StringList como parâmetro. Em seguida, é realizado um laço na StringList, percorrendo cada uma de suas linhas (linha 15), na linha 17 é testado se é encontrada a classe via RTTI através do método FindClass, caso seja encontrado, é utilizado o

método InheritsFrom passando o TfrmModelo como parâmetro. FindClass retorna um ponteiro para a classe, caso seja encontrado. A segunda condição da linha 17 diz que a classe encontrada, além de descender do TfrmModelo, deve ser diferente dela mesma, isso porque não será necessário mapear o formulário de herança. A partir da linha 20, é verificado se não foi encontrado o formulário, assim sendo, é realizado o seu cadastro de forma dinâmica entre as linhas 21 a 28. Observe que para capturar o Caption do formulário, o formulário é criado e destruído em seguida, utilizando a classe TFormClass (linhas 24 a 27). Nas linhas 31 e 32 verificamos se ainda há uma instância não destruída da variável f, destruindo-a caso ainda exista.

Listagem 19. Método ReadFormsAndCreateGrants

```
//Adicione a unit uEnumClasses ao uses
1. procedure TDMPrincipal.ReadFormsAndCreateGrants;
2. var
3.   stList : TStringList;
4.   boApply : Boolean;
5.   i : Integer;
6.   fc : TFormClass;
7.   f : TForm;
8. begin
9.   stList  := TStringList.Create;
10.  boApply := False;
11.  try
12.    cdsFormularios.Close;
13.    cdsFormularios.Open;
14.    EnumClasses(stList);
15.    for i := 0 to stList.Count - 1 do
16.    begin
17.      if ((FindClass(stList[i]).InheritsFrom(FindClass('TfrmModelo')))
       and (UpperCase(stList[i]) <> 'TFRMMODELO')) then
18.      begin
19.        if not cdsFormularios.Locate('NOME_FORMULARIO',
       stList[i], []) 20 then
20.        try
21.          cdsFormularios.Append;
22.          cdsFormulariosIDFORMULARIO.Value := getID('IDFORMULARIO');
23.          cdsFormulariosNOME_FORMULARIO.AsString := stList[i];
24.          fc := TFormClass(FindClass(stList[i]));
25.          f := fc.Create(nil);
26.          cdsFormulariosTITULO_FORMULARIO.AsString := f.Caption;
27.          FreeAndNil(f);
28.          cdsFormularios.Post;
29.          boApply := True;
30.        finally
31.          if Assigned(f) then
32.            FreeAndNil(f);
33.        end;
34.      end;
35.    end;
36.    if boApply then
37.      cdsFormularios.ApplyUpdates(0);
38.    cdsFormularios.Refresh;
39.    CreateGrants;
40.  finally
41.    FreeAndNil(stList);
42.  end;
43. end;
```

Encerrando, caso a variável boApply seja verdadeira, é realizado o ApplyUpdates dos formulários que não estavam cadastrados, e chamada posteriormente o procedure CreateGrants para definir estas permissões ao usuário logado. A destruição da StringList ocorre na linha 41.

Com isso, estamos quase finalizando o Data Module. As implementações finais encontram-se nas **Listagens 20** e **21.** A primeira demonstra os métodos de pesquisa que serão utilizados no cadastro de usuários, já a seguinte demonstra apenas os eventos OnNewRecord dos ClientDataSets, atribuindo os valores das chaves primárias.

Na **Listagem 20** temos a implementação de duas consultas que são praticamente idênticas. Na primeira, entre as linhas 01 e 08 temos simplesmente o fechamento da query, a adição da instrução SQL concatenando o ID do usuário. Já na segunda o processo é o mesmo, porém, é acrescido o nome de usuário. Essas pesquisas serão utilizadas posteriormente quando criarmos o cadastro de usuário.

O código da **Listagem 21** simplesmente alimenta as chaves primárias de cada um dos ClientDataSets.

Listagem 20. Métodos utilizados posteriormente na pesquisa

```
1. procedure TDMPrincipal.SearchUserById(Id: Integer);
2. begin
3.   qryPesquisa.Close;
4.   cdsPesquisa.Close;
5.   qryPesquisa.SQL.Clear;
6.   qryPesquisa.SQL.Text := 'select IDUSUARIO, NOME_USUARIO from USUARIO' +
       ' WHERE IDUSUARIO = ' + Id.ToString();
7.   cdsPesquisa.Open;
8.  end;

9.  procedure TDMPrincipal.SearchUserByUserName(UserName: String);
10. begin
11.   qryPesquisa.Close;
12.   cdsPesquisa.Open;
13.  qryPesquisa.SQL.Clear;
14.  qryPesquisa.SQL.Text := 'select IDUSUARIO, NOME_USUARIO from USUARIO' +
       ' WHERE NOME_USUARIO LIKE %' + QuotedStr(UserName);
15.   cdsPesquisa.Open;
16. end;
```

Listagem 21. Eventos OnNewRecord

```
1. procedure TDMPrincipal.cdsCadEstadoNewRecord(DataSet: TDataSet);
2. begin
3.   cdsCadEstadoIDESTADO.Value := getID('IDESTADO');
4. end;

5. procedure TDMPrincipal.cdsCadUsuarioNewRecord(DataSet: TDataSet);
6. begin
7.   cdsCadUsuarioIDUSUARIO.Value := getID('IDUSUARIO');
8. end;

9.  procedure TDMPrincipal.cdsCadUsuarioPermissaoNewRecord(DataSet:
     TDataSet);
10. begin
11.   cdsCadUsuarioPermissaoIDPERMISSAO_USUARIO.Value :=
       getID('IDPERMISSAO_USUARIO');
12. end;
```

Criando os formulários a partir do modelo

Até aqui já criamos o banco de dados, a herança e a camada de acesso a dados. Agora é hora de criarmos nossos cadastros. Iniciaremos então pelo cadastro mais simples, o de estado, que será utilizado apenas para efeito de exemplo. Sendo assim, crie

um novo formulário a partir do modelo, acessando File->New->Other->Delphi Projects->Inheritable Items->frmModelo, conforme mostra a **Figura 5**.

No formulário que se abre, nomeie-o para frmEstado, salvando sua unit como uEstado.pas. Nele acrescente três Labels e três DBEdits, adicione ao uses do formulário a unit uDMPrincipal e em seguida, configure o dsrCadastro.DataSet apontando para o cdsCadEstado do Data Module. Aponte cada um dos DBEdits para o dsrCadastro e seus respectivos DataFields. Na **Figura 6** temos o layout do cadastro de estados sugerido. Vale lembrar que como este é apenas um exemplo, não desenvolveremos a tela de pesquisa.

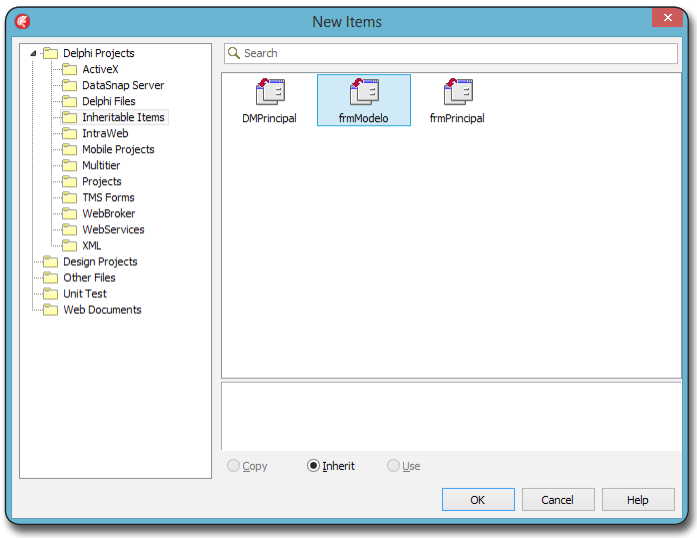


Figura 5. Criação do cadastro a partir do modelo

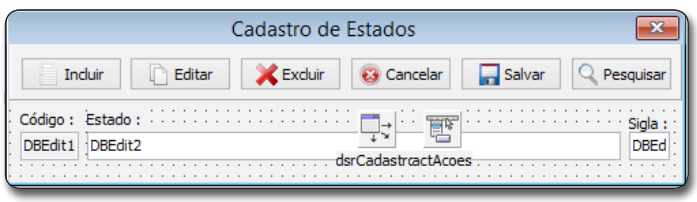


Figura 6. Cadastro de Estados

Para finalizar o cadastro de estados vamos incluir a cláusula RegisterClass ao final do formulário conforme mostrado a seguir:

```
initialization
 RegisterClass(TfrmCadEstado);
```

Essa será a única implementação necessária no formulário, fazendo com que automaticamente seja incluído este formulário tanto na tabela de usuários quanto na tabela de permissões. Passaremos então para o cadastro de usuários.

Criando o cadastro de usuários

Assim como o cadastro de estados, crie um novo formulário a partir do modelo. Renomeie-o para frmCadUsuario e salve a unit como uCadUsuario.pas. Adicione ao novo formulário um PageControl (pclPrincipal) com duas abas (tbsDados e tbsPermissoes) alinhado alClient e um DataSource (dsrUsuarioPermissao). Na primeira aba adicione três Labels e três DBEdits. Adicione o DMPrincipal ao uses deste cadastro. Configure o dsrCadastro apontando para o cdsCadUsuario e o dsrUsuarioPermissao.DataSet setando o cdsCadUsuarioPermissao. Configure os DataFields apontando para os respectivos campos e no DataSource, selecione para isso dsrCadastro. A **Figura 7** mostra a o layout proposto para a primeira aba.

Na segunda página adicionaremos um DBNavigator alinhado alTop e um DBCtrlGrid alinhado alClient. Dentro do DBCtrlGrid adicionaremos um Label, um DBText e sete DBCheckBox. Assim como na primeira aba, faça as devidas ligações, conecte os componentes no dsrUsuarioPermissao e informe seus respectivos DataFields. A **Figura 8** mostra como foi o layout sugerido para a segunda aba.

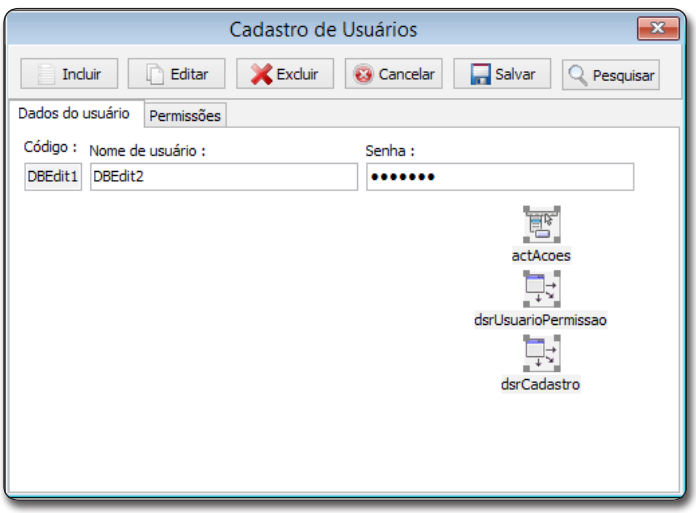


Figura 7. Cadastro de Usuários (aba principal)

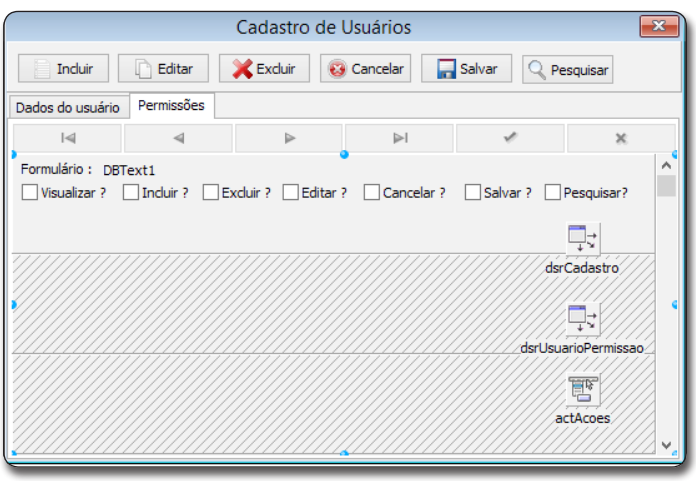


Figura 8. Cadastro de Usuários (aba de permissões)

Antes de passarmos para a codificação do formulário de usuário, criaremos a tela de pesquisa a partir do modelo, sendo assim, crie novamente um formulário a partir do modelo, no entanto,

desta vez será selecionado o frmPesquisa. Altere seu Name para frmPesquisaUsuario, salvando a unit como uPesquisaUsuario.pas. Na propriedade Items do componente cmbParametro, adicione os valores "Código" e "Nome de Usuário". Aponte o dsrPesquisa para o DMPrincipal.cdsPesquisa. A **Figura 9** ilustra como deverá ficar a nossa tela.

Faremos agora as implementações referentes à pesquisa. A **Listagem 22** ilustra as codificações necessárias nesta tela.

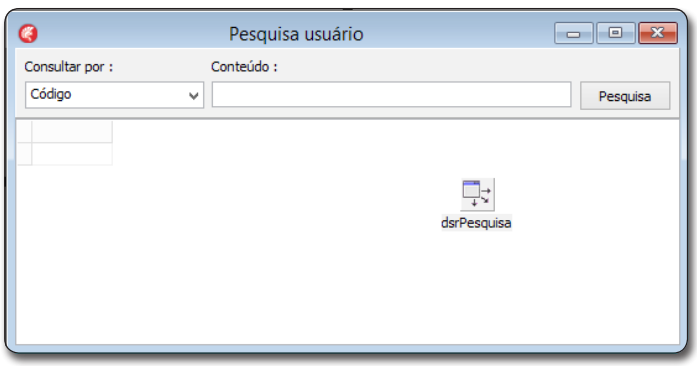


Figura 9. Pesquisa de usuário

Listagem 22. Eventos do formulário de pesquisa de usuário

```
//Adicione o DMPrincipal ao uses do formulário
1. procedure TfrmPesquisaUsuario.btnPesquisaClick(Sender: TObject);
2. begin
3.   inherited;
4.   case cmbParametro.ItemIndex of
5.     0 : DMPrincipal.SearchUserById(StrToInt(edtConteudo.Text));
6.     1 : DMPrincipal.SearchUserByUserName(edtConteudo.Text);
7.   end;
8. end;

9. procedure TfrmPesquisaUsuario.DBGrid1DblClick(Sender: TObject);
10. begin
11.   inherited;
12.   Self.ModalResult := mrOk;
13. end;

14. function TfrmPesquisaUsuario.getSelectedUser: Integer;
15. begin
16.   Result := DMPrincipal.cdsPesquisa.Fields[0].Value;
17. end;
```

Entre as linhas 01 e 08 tem-se o evento OnClick do botão de Pesquisa. Note que simplesmente é verificado o valor selecionado através do ItemIndex do ComboBox. Para a posição zero é utilizado o método SearchUserById, já para a posição um é adotado SearchUserByUserName. Em ambos os casos o conteúdo do Edit é passado como parâmetro. Já entre as linhas 09 a 13 temo o evento OnDoubleClick do DBGrid, que simplesmente atribui o ModalResult do formulário como mrOk (linha 12). Por último, entre as linhas 14 a 17 temos o método getSelectUser que é de escopo público e simplesmente retorna o valor do registro selecionado.

Voltando às implementações do cadastro de usuários, o próximo passo será adicionar o Form de pesquisa ao uses do cadastro. Em seguida partiremos para a parte de mão na massa.

A **Listagem 23** mostra duas simples modificações que faremos, enquanto a **Listagem 24** mostra como será a implementação da pesquisa no ActionList.

Observe na **Listagem 23** que simplesmente é adicionada a linha 03 à ação de incluir do ActionList. Isto para que após a inserção, o PageControl automaticamente volte para a página de dados do cadastro. O mesmo ocorre com a ação de salvar, no entanto, ela vem após a palavra-chave inherited (linha 09). Esta particularidade se dá devido a palavra inherited executar o código do antecessor, ou seja, a implementação realizada no formulário de herança. Por último, temos o RegisterClass ao final do formulário, que simplesmente incluirá o mesmo na lista de Forms mapeados pelo nosso código.

Finalizando as implementações do cadastro de usuários, faremos agora a codificação da ação Pesquisar no ActionList, que encontra-se na **Listagem 24**.

Listagem 23. Implementações e modificações no formulário

```
1. procedure TfrmCadUsuario.actIncluirExecute(Sender: TObject);
2. begin
3.   pcPrincipal.ActivePage := tbsDados;
4.   inherited;
5. end;

6. procedure TfrmCadUsuario.actSalvarExecute(Sender: TObject);
7. begin
8.   inherited;
9.   pcPrincipal.ActivePage := tbsDados;
10. end;

11. initialization
12.   RegisterClass(TfrmCadUsuario);
```

Listagem 24. Ação de pesquisa do ActionList

```
1. procedure TfrmCadUsuario.actPesquisarExecute(Sender: TObject);
2. begin
3.   inherited;
4.   frmPesquisaUsuario := TfrmPesquisaUsuario.Create(Self);
5.   with frmPesquisaUsuario do
6.   try
7.     if ShowModal = mrOk then
8.     begin
9.       DataSetCadastro.Close;
10.      DataSetCadastro.FetchParams;
11.      DataSetCadastro.Params[0].Value := getSelectedUser;
12.      DataSetCadastro.Open;
13.    end;
14.  finally
15.    FreeAndNil(frmPesquisaUsuario);
16.  end;
17. end;
```

Na **Listagem 24** podemos ver um código simples e comum de uma chamada de formulário convencional. Na linha 04 o formulário de pesquisa é criado, a seguir, na linha 07, é aguardado o ModalResult. Note que a condição aguarda o mrOk resultante do duplo clique da Grid na tela de consulta. Vale lembrar que aqueles códigos poderiam ser inseridos no formulário de herança da pesquisa, reduzindo a necessidade de escrevê-lo mais vezes

para outras pesquisas. Entre as linhas 09 e 12, o DataSetCadastro é fechado e o valor selecionado é passado como parâmetro (através do método getSelectedUser, implementado na pesquisa). Por fim, o formulário de pesquisa é destruído e liberado da memória.

Com isso finalizamos o cadastro de usuário. Para a finalização do nosso protótipo, basta agora a criação do Login de usuário e do menu principal da aplicação, que serão demonstrados a seguir.

Criando o Login de usuário

Finalizado o cadastro, faremos a tela de autenticação do usuário. Para isso, crie um novo formulário e nomeie-o como frmLogin, salvando sua unit como uLogin.pas. No novo formulário adicione dois Labels, dois Edits (edtUsuario, edtSenha) e um BitBtn (btnLogar), conforme mostra a **Figura 10.** Sua única codificação será no evento OnClick do botão, conforme o código da **Listagem 25**.

Observe que simplesmente é passado para a função CanLogin do DMPrincpal os valores contidos nos Edits edtUsuario e edtSenha. Caso a função retorne verdadeiro, o ModalResult do formulário será mrOk, permitindo a entrada na aplicação, caso contrário, é exibida a mensagem de usuário ou senha inválidos!

Listagem 25. Evento OnClick do botão btnLogar

```
1. //adicione o DMPrincipal ao uses do formulário
2. if DMPrincipal.CanLogin(edtUsuario.Text, edtSenha.Text) then
3.   Self.ModalResult := mrOk
4. else
5.   MessageDlg('Usuário ou senha inválidos!', mtConfirmation, [mbOk], 0);
```

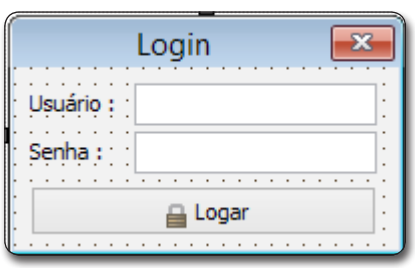


Figura 10. Formulário de Login do usuário

Criando o formulário principal da aplicação

Para finalizar nossa aplicação, criaremos o formulário de Menu principal. Este será o responsável tanto por chamar a tela de Login de usuário, como concentrar os formulários do sistema. Crie um novo Form nomeando-o como frmPrincipal e salvando sua unit como uPrincipal.pas. A seguir, adicione um componente MainMenu e, com um duplo clique, crie os itens conforme mostrado na **Figura 11**.

Feita a inserção dos respectivos itens de menu, faremos a chamada do formulário de Login. Adicione no uses do frmPrincipal o DMPrincipal, o frmLogin, o frmCadUsuario e o frmCadEstado. O primeiro passo será executar a chamada do Login que ocorrerá por meio do evento OnShow do formulário, implementado na **Listagem 26.**

No evento OnShow temos uma chamada semelhante a qualquer chamada de formulário. Na linha 03 é criado o Login, a seguir, na linha 05, é aguardado o ModalResult mrOk (retornado a partir da confirmação de sucesso ao se logar). Caso seja retornado, por exemplo, mrCancel (ação de fechar do Form), a linha 06 executará o método Application.Terminate, encerrando a aplicação, caso contrário, o método ReadFormsAndCreateGrants é chamado (linha 08), fazendo com que todas as permissões sejam definidas e aplicadas para o usuário logado.

Com isso, basta realizarmos as chamadas aos formulários de cadastro e usuário. Como a chamada de ambos é idêntica, diferenciando apenas os formulários e o DataSet, a **Listagem 27** ilustrará apenas a chamada do cadastro de estados.

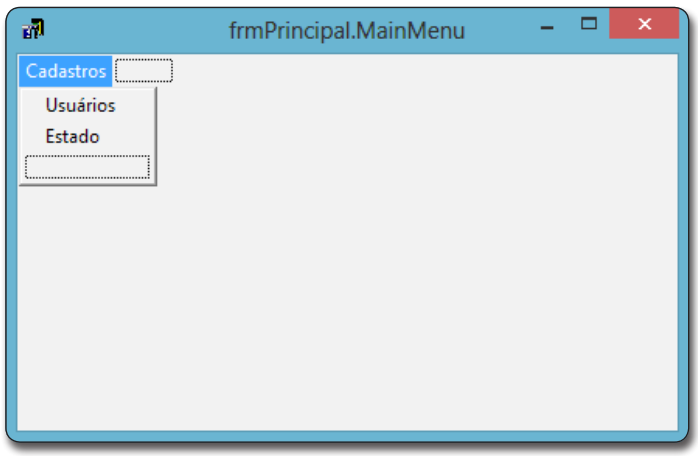


Figura 11. Definindo o MainMenu

Listagem 26. Evento OnShow do formulário principal

```
1. procedure TfrmPrincipal.FormShow(Sender: TObject);
2. begin
3.   frmLogin := TfrmLogin.Create(Self);
4.   try
5.     if frmLogin.ShowModal <> mrOk then
6.       Application.Terminate
7.     else
8.       DMPrincipal.ReadFormsAndCreateGrants;
9.   finally
10.    FreeAndNil(frmLogin);
11.  end;
12. end;
```

Listagem 27. Chamada do cadastro de estados

```
1. procedure TfrmPrincipal.Estado1Click(Sender: TObject);
2. begin
3.   if CanCreateForms(TfrmCadEstado.ClassName) then
4.   begin
5.     frmCadEstado := TfrmCadEstado.Create(Self);
6.     with frmCadEstado do
7.     try
8.       DataSetCadastro  := DMPrincipal.cdsCadEstado;
9.       DataSetPermissao := DMPrincipal.cdsConfiguraPermissao;
10.      ShowModal;
11.    finally
12.      FreeAndNil(frmCadEstado);
13.    end;
14.  end;
15. end;
```

Podemos ver o uso da função CanCreateForms, que recebe o nome da classe como parâmetro. Na linha 03 foi utilizado o método ClassName no lugar da String fixa, isso porque caso você altere o formulário, ficará mais fácil mapear o local de mudança e corrigi-lo. Caso o retorno seja verdadeiro, simplesmente criamos o cadastro de estados (linha 05), atribuímos os DataSetCadastro e DataSetPermissao (linhas 08 e 09) e exibimos a tela (linha 10), assim como nas demais chamadas, o formulário é destruído posteriormente (linha 12).

Testando a aplicação

Para realizar um teste na aplicação, inserimos manualmente um usuário e uma senha no banco de dados e em seguida efetuamos o Login do mesmo. Abra as tabelas de formulários e de permissão e perceba que os dados foram alimentados automaticamente. Em seguida, acesse o formulário de usuários e consulte o usuário cadastrado. Teremos a possibilidade de alterar suas permissões de acordo com cada formulário do sistema. Em nosso exemplo, desmarcaremos primeiramente a permissão de visualizar do cadastro de estados. Saia do sistema e entre novamente tentando realizar o acesso, o resultado você observa nas **Figura 12, 13 e 14.**

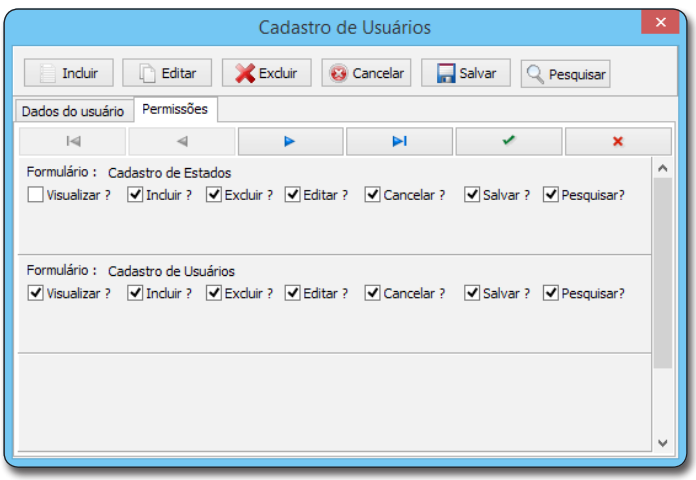


Figura 12. Aplicação em execução: restrição de permissão de visualização do cadastro de estado

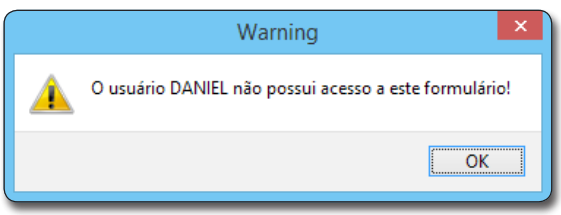


Figura 13. Restrição de acesso desativando o botão de visualizar

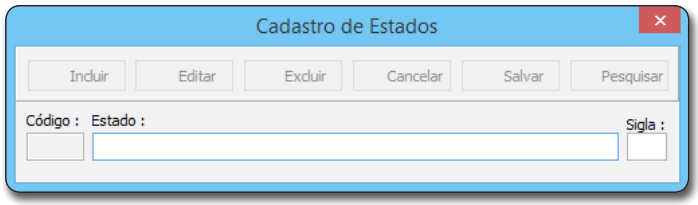


Figura 14. Restrição de acesso referente aos botões

Como sabemos, há sempre maneiras de se implementar a segurança utilizando os componentes de terceiros, e obviamente, essas maneiras oferecem bastante recursos interessantes e não só desvantagens, no entanto, criar seus próprios recursos é sempre bem mais interessante. Vale lembrar que os conceitos vistos aqui são superficiais, permitindo ir muito além da ideia de simplesmente mapear formulários. Utilizando os recursos de RTTI, por exemplo, podemos expandir a ideia e mapear os controles das próprias telas, permitindo assim restringir o acesso não só a formulários, mas sim a relatórios e campos, ocultando ou desabilitando e etc. Fica aqui uma sugestão de como implementar os controles de usuário em suas aplicações além de uma ideia para que você ampliar o tema abordado neste artigo.

Autor

Daniel Sobrinho Laporte

Colunista e editor técnico da revista Clube Delphi. Graduado em Sistemas de informação, pós-graduado em administração de bancos de dados e pós-graduando em Engenharia de Software com métodos ágeis. Atualmente é analista de sistemas e desenvolvedor na empresa Orgsystem Software. Certificado Delphi Developer, trabalha com o IDE desde 2005 adotando arquiteturas cliente-servidor e multicamadas com os mais variados bancos de dados.

Você gostou deste artigo?

Dê seu voto em ***www.devmedia.com.br/clubedelphi/feedback***

Ajude-nos a manter a qualidade da revista!

MVVM para programadores Delphi

Descomplique o uso deste padrão

Apesar de não tão recorrente, o tema MVVM já foi abordado anteriormente em algumas ocasiões aqui mesmo nesta revista. Assim como esperado, tais abordagens tiveram como foco a apresentação da verdadeira essência deste padrão, conceituando o leitor numa tomada bastante abrangente, sem restrições. Como fruto disso, a plena assimilação dos conceitos provia a empregabilidade do MVVM em diversos cenários, independente de linguagem ou tecnologia. Apesar de eficaz, o uso de padrões deste tipo acaba por imprimir, naturalmente, diversas barreiras ao desenvolvedor Delphi tradicional. Aqui, considera-se tradicional o programador que faz uso efetivo do poder RAD provido pela ferramenta, por meio do uso dos componentes disponibilizados, codificação de seus eventos, entre outros hábitos. A partir disso, nota-se que grande parte da comunidade acaba por se encaixar neste agrupamento, que não vê nos padrões de projeto, tal qual o MVVM, algo essencial ou de importância para seu uso habitual. Logo, por esta razão, o entendimento do assunto torna-se bastante prejudicado.

Diante disso o presente artigo vai na contramão do paradigma "essencial", uma vez que se restringe ao contexto Delphi, como linguagem e tecnologia, provendo o assunto de uma forma singular, por meio do bordão "MVVM para programadores Delphi". Tudo isso com o intuito de simplificar tanto a assimilação do tema, quanto o emprego prático do mesmo, sempre tendo como base a possível visão de um programador desta comunidade. Por se tratar de um assunto não tão comum na comunidade Delphi, neste início de abordagem, torna-se imprescindível a explanação de toda a parte conceitual relacionada ao tema.

Uso da POO no desenvolvimento Delphi

No ramo da programação, a Orientação a Objetos, ou simplesmente OO, é um paradigma que estabelece a projeção e construção de softwares a partir de um agrupamento de unidades sistêmicas que interagem entre si, na busca por atender um fim em comum. Cada unidade desta é então tratada como "objeto", oriundo de um elemento mais abrangente denominado "classe". Neste cenário, uma aplicação pode conter um conjunto de classes implementadas, cada qual determinando os eventuais comportamentos (por meio de métodos) e estados (por meio de atributos) de um tipo especificado de objeto. Logo, toda essa relação entre classes e objetos é comumente resumida na seguinte afirmação: "um objeto nada mais é do que uma instância de uma classe".

O Delphi, sendo uma linguagem OO, também oferece a seus desenvolvedores recursos suficientes para a construção de softwares apoiados no paradigma da POO. Apesar disso, tendo em mente um desenvolvimento mais "tradicional", baseado no arrastar e soltar de componentes para formulários e codificação de eventos, todo o conceito da POO fica restrito apenas aos bastidores da construção. Em outras palavras, isso significa dizer que mesmo nessas situações, de forma indireta, o desenvolvedor acaba por fazer uso da orientação a objetos em seu processo desenvolvimento. Como exemplo, um componente TButton arrastado para um Form fundamenta um objeto, uma vez que formaliza uma instância da classe TButton. Em contraponto, a essência mais purista da POO não é realmente explorada nessas construções, favorecendo o desenvolvedor a abdicar

Fique por Dentro

Em termos gerais, o desenvolvimento efetivo de um software, independente da linguagem de programação utilizada, pode ser apoiado no entendimento e uso de conceitos concretos oriundos de determinados padrões exclusivos deste segmento. No meio, tais padrões são tratados como Padrões de Projeto que, de uma forma ampla, caracterizam uma forma segura de desenvolvimento pautada em situações já experimentadas no mundo real, o que tende a evitar problemas ocasionais, contribuindo para um processo de desenvolvimento bastante eficaz. Em meio aos diversos padrões existentes, o MVVM é um padrão que trata da arquitetura do software, incidindo diretamente em seu aspecto construtivo. Assim, o presente artigo tem como foco sua explanação e empregabilidade no contexto do Delphi, numa abordagem essencialmente direcionada a este tipo de desenvolvedor.

da criação de suas próprias classes e, em consequência, seus próprios objetos.

Em vista disso, o uso de padrões de projeto, que por sua própria natureza já são enraizados na POO, tende a se tornar algo dificultoso para um desenvolvedor Delphi "tradicional". Isto porque, o mesmo acaba por não conseguir empregar uma técnica que irá lidar com classes e objetos, cujas atividades não fazem parte de sua rotina habitual. Sendo assim, a intenção deste artigo é dar fim a este estigma, tentando aproximar o contexto do Delphi ao da POO e, por conseguinte, do uso do padrão MVVM, deixando de lado, para isso, certos aspectos "puristas" tão presentes em abordagens rotineiras sobre o tema.

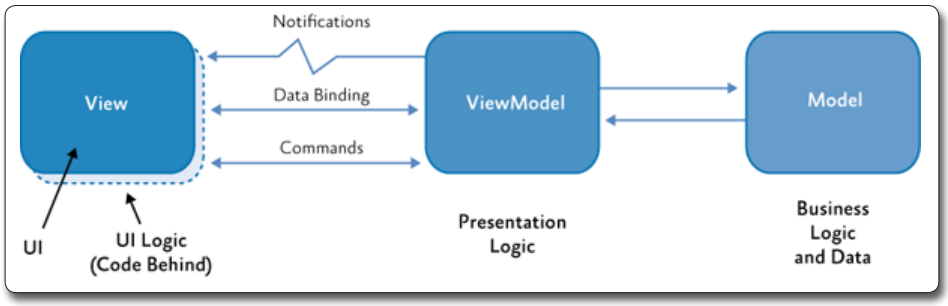


Figura 1. Ilustrativo do padrão MVVM

Aspectos conceituais do MVVM

MVVM é o acrônimo para Model-View-ViewModel, cuja definição técnica se faz por um padrão de projeto cuja finalidade principal, em termos gerais, é favorecer a segmentação do contexto da aplicação, dividindo-a em partes bem estabelecidas. Tal separação tende a ser clara o suficiente a ponto de contribuir em aspectos que vão da elaboração da aplicação, passando por sua manutenção e eventual expansão. Até este ponto, o MVVM se assemelha muito com outros padrões já conhecidos, tais como os já habituais MVC (**BOX 1**) e MVP, todavia, assim como estes, possui suas próprias particularidades que acabam por formalizar sua real intenção. Assim, tal intenção pode ser resumida na busca pela separação de três elementos que, por vezes, tendem a estarem associados: lógica de negócio (business logic), lógica de apresentação (presentation logic) e interface de usuário (UI). Num cenário recente, o MVVM sempre esteve associado a algumas das tecnologias de desenvolvimento da Microsoft, fundamentalmente àquelas que se baseiam no uso de XAML (**BOX 2**), fazendo com que o padrão tirasse proveito de aspectos específicos da tecnologia, tais como Commands, Data Binding e Behaviors. Logo, a partir deste ponto de vista, grande parte das literaturas atuais estabelece o padrão tal como ilustrado na **Figura 1**. Pelo fato maior de envolver diversos aspectos não pertinentes ao Delphi, esse tipo de conceituação tende a dificultar o entendimento por parte dos desenvolvedores da comunidade. Assim, num primeiro momento, talvez a assimilação ideal se dê pelo afunilamento de conceitos, o que resulta em algo próximo ao que é mostrado na **Figura 2**.

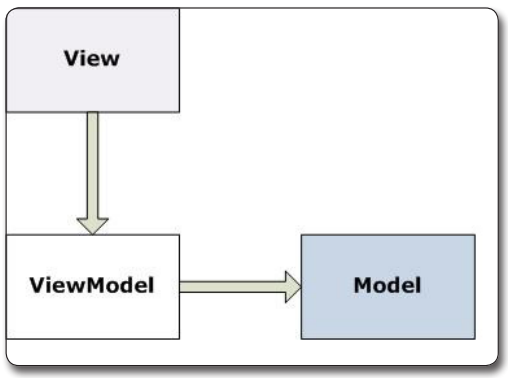


Figura 2. Ilustrativo enxuto do padrão MVVM

BOX 1. MVC e MVP

Model-View-Controller ou simplesmente MVC, é um padrão de projeto relacionado à arquitetura de software, que segmenta a aplicação em três partes bem definidas e interligadas. A primeira, denominada Model (M), realiza as tratativas de dados e regras de negócio da aplicação. A segunda, tratada como View (V), nada mais é do que qualquer saída que venha a representar os dados do aplicativo, lugar este que é o ponto visual entre software e usuário. Por fim, a parte nomeada como Controller (C) é responsável por ser a mediadora das entradas do usuário, repassando esta interação de forma direta ou indireta, tanto para Model quanto para View. O MVP é uma variação do MVC e representa Model-View-Presenter. Pode-se dizer que o MVP é um MVC especializado para visualização e a diferença entre si são mais conceituais. Ao imaginar um cadastro de clientes, o Model é a coleção de objetos TCliente que são manipulados no formulário, o formulário por sua vez é a View. O presenter é responsável por interceptar os eventos gerados na View, com a finalidade de controlar a lógica de apresentação. Com essa definição se vê a semelhança, contudo existem alguns pontos de diferença: no MVC a View se comunica (envia mensagens) com o Model, já no MVP a View pode conhecer o Model mas não envia mensagens para ele, isso é feito através do Presenter.

BOX 2. XAML

XAML é o acrônimo para o termo em inglês Extensible Application Markup Language. Conceitualmente, pode ser definida como sendo uma linguagem baseada em XML, utilizada na definição de interfaces gráficas de usuário para aplicações relacionadas a determinadas tecnologias Microsoft, tais como WPF e Silverlight.

View

No MVVM, a letra V representa sua entidade mais simples, que é a camada de Visão, comumente tratada como View, seu termo análogo em inglês. Em suma, trata-se da parte visual da aplicação, tal qual uma simples página ou janela, que acaba por fundamentar uma interface de usuário. Assim, como esta própria definição sugere, atua como um ponto de contato visual com o utilizador do software (usuário). Logo, a responsabilidade de uma View, neste cenário, deve se restringir a este atributo, resultando na definição da estrutura e aparência do layout em tela, ou seja, aquilo que o usuário final irá enxergar. De fato, muito pouco código deverá existir no code-behind de uma classe de Visão, o que inclui apenas algumas eventuais tratativas da lógica pertinente à própria interface visual.

Model

Em contraponto à View, que se mostra como uma simples entidade visual, a letra M do padrão faz referência a um segmento não visual denominado Modelo (Model), e que acaba por tratar de boa parte da complexidade do software. Isto porque ele encapsula toda a parte de dados e lógica de negócios envolvida, contextualizando o modelo de domínio do lado cliente da aplicação. Dessa forma, é comum o Modelo prover apoio a artefatos, tais como restrições e validações, que venham a contribuir para a ideal consistência da lógica e dos dados relacionados. Além disso, a busca pela máxima generalização durante a implementação dessas regras torna-se um ponto chave importante neste cenário, uma vez que favorece sua reutilização. Em outras palavras, um modelo bem definido nestes critérios tende a estar disponível para uso em diversas situações, não ficando restrito a um único projeto específico. Por tudo isso, é desaconselhável a introdução de tratativas e comportamentos muito específicos em seu corpo.

ViewModel

A ViewModel estabelece um ponto intermediário entre as partes View e Model do padrão, cuidando especificamente da lógica da Visão. Adicionalmente, interage com o Modelo a fim de promover a disponibilidade de métodos de negócio e dados para a View. Desta maneira, torna os pontos extremos da aplicação, representados por View e Model, independentes um do outro. Numa exemplificação típica, torna-se então possível a existência de diferentes variações de Visão para um mesmo Modelo.

Aspectos práticos do MVVM

Na prática, a adoção do padrão MVVM se reflete essencialmente no uso de três classes distintas, que acabam por representar cada um dos elementos citados anteriormente, que compõem sua estrutura:

- Model: a classe Model diz respeito ao modelo da aplicação, encapsulando toda a lógica de negócio (business) e dados (data) envolvida;
- Visão: a classe Visão é relativa à interface de usuário, dessa forma, ela ficará responsável por encapsular toda a lógica pertinente à UI;
- ViewModel: a classe ViewModel pode ser considerada o artefato peculiar do padrão MVVM que o diferencia dos demais padrões existentes, uma vez que cuida dos aspectos de estado e lógica de apresentação pertinentes.

Tendo esta segmentação em mente, é possível afirmar que a chave para se utilizar o referido padrão de forma efetiva no cenário Delphi se reflete primeiramente na ideal compreensão da interação entre as classes. Posteriormente, o ponto final, e talvez o de maior complexidade, se dá por encontrar a forma adequada de se levar e distribuir o código da aplicação para as classes adequadas. Com a conceituação básica do padrão já em mente, a partir deste ponto, a intenção do artigo se volta ao entendimento da introdução do padrão MVVM no desenvolvimento de aplicações Delphi de banco de dados tidas como "tradicionais", que é a realidade de grande parte da comunidade. Por conseguinte, o objetivo final desta abordagem é demonstrar, na prática, uma forma simples de se exibir os dados da tabela Customer do banco de dados na interface gráfica da aplicação, mediante o uso otimizado deste padrão.

A classe Model

No âmbito prático, já de início é importante ressaltar que a essência do MVVM prega que um Modelo não deve fazer referência direta a nenhuma classe View ou ViewModel, abdicando assim de qualquer dependência no que tange a forma com que estas foram implementadas. No quesito prático, em linhas gerais, a implementação de uma classe Model tende a elucidar uma construção envolvendo puramente código, sem qualquer aspecto visual. Em linguagens como Java e C#, apenas para citar duas, esse tipo de abordagem, envolvendo a codificação manual de classes, é algo bastante trivial e natural. A justificativa inicial para isso pode ser resumida na própria característica construtiva de ambas, que não possuem um fator RAD tão determinante quanto o Delphi. Isso faz com que o desenvolvedor escreva mais código e se familiarize mais rapidamente com o paradigma da POO. Logo, é possível afirmar que a construção de uma classe Model no Delphi, envolvendo puramente código, é algo plenamente possível, porém não tão habitual. Conforme já mencionado até aqui, no MVVM, o Model fica responsável, entre outras coisas, pela parte de dados da aplicação. Em vista disso, a **Listagem 1** mostra a definição de uma classe ilustrativa denominada TCustomer, que poderia ser utilizada para representar uma eventual entidade Cliente da aplicação, englobando então suas informações e, consequentemente, seus dados. A fim de favorecer o entendimento, a própria classe, incluindo seus atributos, é parcialmente baseada na tabela

Listagem 1. Classe Model (TCustomer)

```
TCustomerModel = class
 private
  FCustNo: Integer;
  FAddressLine2: string;
  FCustomer: string;
  FAddressLine1: string;
  FCountry: string;
  FPhoneNo: string;
  FCity: string;
  procedure SetAddressLine1(const Value: string);
  procedure SetAddressLine2(const Value: string);
  procedure SetCity(const Value: string);
  procedure SetCountry(const Value: string);
  procedure SetCustNo(const Value: Integer);
  procedure SetCustomer(const Value: string);
  procedure SetPhoneNo(const Value: string);
 public
  property CustNo: Integer read FCustNo write SetCustNo;
  property Customer: string read FCustomer write SetCustomer;
  property PhoneNo: string read FPhoneNo write SetPhoneNo;
  property AddressLine1: string read FAddressLine1 write SetAddressLine1;
  property AddressLine2: string read FAddressLine2 write SetAddressLine2;
  property City: string read FCity write SetCity;
  property Country: string read FCountry write SetCountry;
end;
```

Customer (**Figura 3**) do tradicional banco de exemplo EMPLOYEE. FDB que acompanha a instalação de qualquer versão do Firebird (para este artigo, foi usada a versão 2.5).

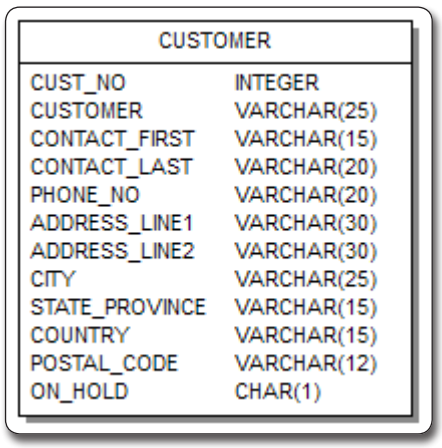


Figura 3. Tabela Customer

Ainda de acordo com a classe apresentada, fica fácil supor que este tipo de construção não é a mais habitual no cenário Delphi. Indo além, para aplicações de banco de dados, toda esta definição é comumente substituída pelo simples uso de um componente de dados (DataSet), tal qual um TSQLQuery (do dbExpress), TFDQuery (do FireDAC) entre outros do gênero, destacando todo o poder RAD do Delphi. Logo, a fim de se manter a característica de desenvolvimento já recorrente à comunidade, a classe Model mostrada poderia, por exemplo, ser substituída pelo uso de um TDataModule. Nele seriam definidas tanto a parte de dados quanto a lógica de negócio da aplicação, de forma a atender as diretrizes do padrão.

Em projetos como esse, com relação à parte de dados, uma boa prática se dá pelo isolamento do ponto de tratamento da conexão com o banco de dados. Em termos RAD, isso poderia ser brevemente feito pelo direcionamento do componente que efetivamente realiza a conexão da aplicação com o banco, para um DataModule à parte. De forma a ilustrar isso, a **Figura 4** mostra tal definição para um cenário onde o FireDAC foi a biblioteca de acesso a dados escolhida para o projeto sendo, portanto, utilizado o componente TFDConnection.

Tendo em vista o Modelo mostrado anteriormente, sua representação RAD é elucidada na **Figura 5**. Em termos de código, essa abordagem torna a classe Model mais enxuta, conforme pode ser visto na **Listagem 2**, uma vez que toda sua parte informativa se concentra agora em um componente não-visual. As "properties" anteriormente existentes são agora representadas por campos (Fields) do DataSet. De forma complementar, a **Listagem 3** exibe o trecho principal do código DFM do componente, que inclui a declaração da instrução SQL utilizada.

Por fim, tendo em vista toda a tratativa de dados aplicada, bem como a não dependência com qualquer outra classe pertinente ao padrão, a conclusão exemplificativa do Modelo poderia ser feita pela adição de um método de negócio. Sem delongas, a adição de um método simples, que efetivamente retorna os "clientes" da

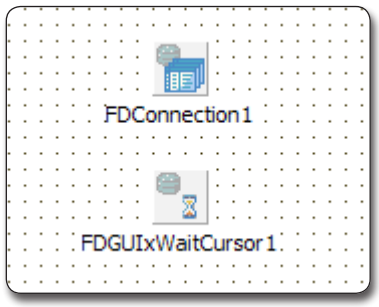


Figura 4. DataModule de conexão

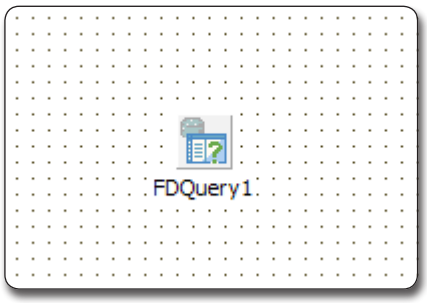


Figura 5. Model

Listagem 2. Nova classe Model (TCustomer)

```
TCustomerModel = class(TDataModule)
  FDQuery1: TFDQuery;
  FDQuery1CUST_NO: TIntegerField;
  FDQuery1CUSTOMER: TStringField;
  FDQuery1PHONE_NO: TStringField;
  FDQuery1ADDRESS_LINE1: TStringField;
  FDQuery1ADDRESS_LINE2: TStringField;
  FDQuery1CITY: TStringField;
  FDQuery1COUNTRY: TStringField;
 end;
```

Listagem 3. Código DFM do componente DataSet

```
object CustomerModel: TCustomerModel
 ...
 object FDQuery1: TFDQuery
  Connection = DataModule2.FDConnection1
  SQL.Strings = (

    'select Cust_No, Customer, Phone_No, Address_Line1, Address_Line2' +
    ', City, Country from Customer')
  ...
 end
end
```

aplicação, cujas informações proveem de uma tabela do banco de dados. Uma vez que se está trabalhando com DataSets, essa tarefa se torna facilitada, conforme mostra a **Listagem 4**, que sumariza o código complementar inserido à classe TCustomerModel. Novamente como forma de se aplicar boas práticas, o referido método foi atrelado à definição de uma property, que poderá então ser utilizada por outras partes do sistema que venham a fazer uso desta classe Model.

A classe ViewModel

Mediante a segmentação estabelecida pelo padrão, View e Model são extremidades singulares que se interconectam por meio de uma classe intermediária denominada ViewModel. Na prática, ela então interage com o Modelo invocando métodos e recuperando dados, tornando-os linearmente disponíveis para a Visão. Logo, para a definição da classe ViewModel, fica implícita a necessidade por uma referência direta à classe Model. De forma breve, isso pode ser feito, por exemplo, pela simples inclusão de um atributo do referido tipo à classe. Por conseguinte, de posse do Modelo ilustrativo definido anteriormente, o provimento de uma classe ViewModel para ele é mostrado na **Listagem 5**.

Listagem 4. Nova classe Model (TCustomer)

```
unit UCustomerModel;
...
TCustomerModel = class(TDataModule)
  ...
 private
  function GetCustomers: TDataSet;
 public
  property Customers: TDataSet read GetCustomers;
 end;

//Implementação do método GetCustomers,
function TCustomerModel.GetCustomers: TDataSet;
begin
 Result := FDQuery1;
end;
```

Listagem 5. Classe ViewModel – Declaração

```
unit UCustomerViewModel;

interface

uses
 Data.DB, UCustomerModel;

type
 TCustomerViewModel = class
 private
  FModel: TCustomerModel;
  function GetCustomerDataSet: TDataSet;
 public
  constructor Create;
  destructor Destroy; override;

  property CustomerDataSet: TDataSet read GetCustomerDataSet;
 end;
...
```

Nesta exemplificação, em suma, o objetivo principal da ViewModel é fornecer meios para que o conjunto de dados presente no modelo esteja posteriormente acessível à View. Para isso foi então declarada uma propriedade na classe, denominada CustomerDataSet, que essencialmente irá armazenar o conjunto de dados provindo do Modelo. Adicionalmente, FModel é o campo que faz referência ao Modelo associado, neste caso, TCustomerModel. No Delphi, para que esta referência se torne possível, é exigida a inclusão da unit onde o referido tipo está declarado (neste caso UCustomerModel) na seção uses da unit atual. Em complemento, a **Listagem 6** exibe a implementação de cada ponto declarativo da classe TCustomerViewModel.

Listagem 6. Classe ViewModel – Implementação

```
constructor TCustomerViewModel.Create;
begin
 inherited Create;
 FModel := TCustomerModel.Create(nil);
end;

destructor TCustomerViewModel.Destroy;
begin
 FModel.Free;
 inherited;
end;

function TCustomerViewModel.GetCustomerDataSet: TDataSet;
begin
 Result := FModel.Customers;
end;
```

Nesta implementação, no construtor da classe ViewModel é adicionada a criação da classe do modelo relacionado. Pelo fato do Delphi não contemplar de um Garbage Collector, recurso este que se encarrega de eliminar referência de objetos não mais utilizados no âmbito da aplicação, é codificado no método destrutor da classe ViewModel a liberação do objeto Model inicialmente criado, por meio de uma chamada ao método Free. Por fim, GetCustomerDataSet é o método de leitura utilizado pela property CustomerDataSet, que recupera o conjunto de dados do modelo. Para este caso, é feita uma chamada à propriedade Customers de TCustomerModel.

A classe View

Conforme já citado anteriormente, todas as tratativas relacionadas ao comportamento visual de uma interface de usuário da aplicação estarão relacionadas a uma classe de Visão. Em suma, essas transições comportamentais que incidem sobre a View são oriundas essencialmente da própria interação com o usuário ou da mudança de estado de sua classe co-relata (ViewModel). No cenário estabelecido pelo padrão MVVM, a View não possui interdependência alguma com o Model o que, em outras palavras, significa dizer que uma Visão não necessita tomar conhecimento do Modelo associado. Transpondo tal afirmação para o contexto da linguagem Delphi, resulta na não adição da unit onde está declarada a classe Model na seção uses da unit da View. Toda essa explicação vem para explicitar então a impossibilidade de uma eventual ligação direta entre algum controle visual determinado na Visão com o DataSet presente no Modelo, tal como já é habitual em um desenvolvimento tradicional de aplicações envolvendo dados no Delphi. A partir disso, determinadas características habituais já não se fazem mais necessárias aqui. O exemplo maior

se dá com relação ao uso de componentes Data-Aware, popularmente tratados como "controles DB" pela comunidade, e que neste paradigma são então substituídos por controles sem ligação direta a banco de dados (Ex: TStringGrid, TEdit, TComboBox, etc.).

Diante disso, sua definição se torna bastante natural e livre. No âmbito do Delphi, isso acaba por envolver então desde uma interface texto, para o caso de aplicações Console, até mesmo UI mais elaboradas, em se tratando de aplicações VCL e FMX. De forma ilustrativa, a seguir, na **Figura 6** é mostrada uma sugestão de View básica, fundamentada no uso da VCL, pressupondo-se então o envolvimento de um projeto do tipo VCL Forms Application. A UI definida tem como base o componente TStringGrid (grupo Additional da Tool Pallete) que se mostra como um controle visual voltado à listagem de dados. Por esta razão, na atual conjuntura, ele é o elemento tido como ideal para substituir o "famoso" TDBGrid. Um detalhe importante a ser mencionado neste ponto está relacionado ao uso de tecnologias de apoio que contribuem diretamente (podendo ser consideradas essenciais) para a boa utilização e aplicação do padrão. Neste caso, todo esse apoio vem de uma tecnologia nativa, o Live Bindings, refletida em dois componentes:

• TBindNavigator: componente visual similar ao TDBNavigator da VCL. Se liga a outros componentes por meio de LiveBindings,

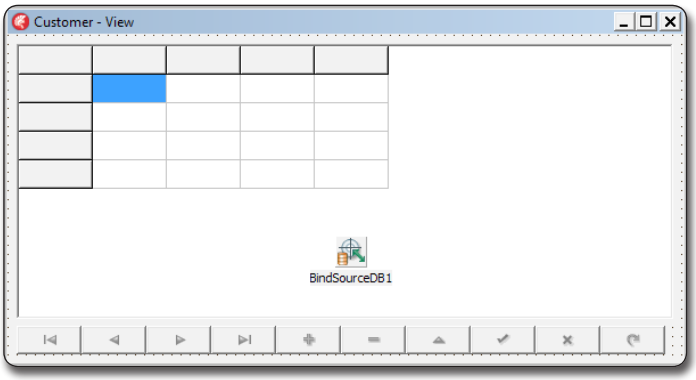


Figura 6. View (VCL)

possibilitando a manipulação sobre os dados envolvidos, o que resulta essencialmente nos processos de navegação e edição das informações. Diante disso, Insert (inserir), Delete (excluir), First (primeiro), Next (próximo) e Edit (editar) são algumas de suas ações disponíveis;

• TBindSourceDB: componente não-visual que se liga a uma fonte de dados especificada a fim de tornar seus dados disponíveis ao contexto de outros componentes, acessíveis por meio de LiveBindings.

CURSOS ONLINE

A Revista SQL Magazine oferece aos seus assinantes uma série de Cursos Online de alto padrão de qualidade.

Feita para Desenvolvedores de Software e DBAs
SQL magazine

CONHEÇA OS CURSOS MAIS RECENTES:

- Cursos: Curso de noSQL (Redis) com Java
- Desenvolvimento para SQL Server com .NET
- Curso PostgreSQL - Treinamento de banco de dados (Curso Básico)

Para mais informações :
http://www.devmedia.com.br/cursos/banco-de-dados
(21) 3382-5038

DEVMEDIA

No caso do componente de navegação (TBindNavigator), apesar de o mesmo estar disponível a partir do grupo LiveBindings da Tool Palette do IDE, seu uso está tão atrelado à presença de um elemento de BindSource (TBindSourceDB e TBindSourceDBX) que sua inclusão pode ser feita por meio do próprio componente deste tipo. Isso ocorre pelo acionamento de sua opção de contexto "Add Navigator", conforme mostra a **Figura 7**.

Em complemento, como o importante aqui é a determinação da classe de Visão propriamente dita, a **Listagem 7** mostra o código associado a este Design.

Na seção private do código apresentado, nota-se então a presença de um Field (campo) que faz referência à classe ViewModel (TCustomerViewModel) desta abordagem. Logo, na seção uses desta unit é declarada a unidade relacionada (UCustomerViewModel) a este tipo. Por meio disso é que a View poderá exibir os dados provindos do Modelo. Dito isto, a **Listagem 8** exibe de forma concreta tal fundamento.

Listagem 7. Declaração da classe View

```
unit UCustomerView;
...
uses
  UCustomerViewModel...;
...
TTCustomerView = class(TForm)
  StringGrid1: TStringGrid;
  BindSourceDB1: TBindSourceDB;
  NavigatorBindSourceDB1: TBindNavigator;
private
  FViewModel: TCustomerViewModel;
end;
```

Listagem 8. Implementação da classe View

```
procedure TTCustomerView.FormCreate(Sender: TObject);
begin
FViewModel := TCustomerViewModel.Create;
BindSourceDB1.DataSet := FViewModel.CustomerDataSet;
end;

procedure TTCustomerView.FormDestroy(Sender: TObject);
begin
FViewModel.Free;
end;
```

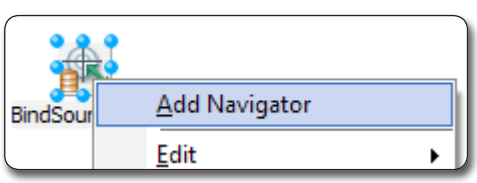


Figura 7. Opção Add Navigator do componente TBindSourceDB

De forma prática, no OnCreate do formulário (View) é então criada uma instância da classe ViewModel, atribuída ao campo FViewModel. Na sequência, de posse deste objeto, torna-se possível o acesso aos dados do Modelo por meio da leitura da propriedade CustomerDataSet da ViewModel. Esta é então atribuída como sendo a fonte de dados do elemento de Binding presente no contexto (componente TBindSourceDB). Com isso, a UI torna-se capaz de exibir as informações oriundas da tabela Customer do banco de dados, mediante as diretrizes do padrão MVVM. Ao final, no OnDestroy do Form, libera-se o objeto ViewModel anteriormente criado por meio do método Free.

Em termos de código, é possível afirmar que a aplicação do padrão MVVM está concluída, uma vez que sua essência foi estritamente respeitada em todas as etapas construtivas. Todavia, ainda neste cenário, um último ajuste se faz necessário, referente à ligação que deve existir entre o elemento TBindSourceDB (BindSourceDB1) e o controle TStringGrid (StringGrid1). Assim como ocorre e é habitual no desenvolvimento tradicional de aplicações de banco de dados, essa interconexão se faz necessária para que os registros relacionados ao componente de dados possam ser idealmente visualizados no componente visual. Ainda diante deste comparativo, aqui não se tem a presença de um TDataSource ou algo do gênero, uma vez que não há envolvimento de controles Data-Aware, assim, ao invés disso, a ligação desejada se dá pelo uso de LiveBindings. Uma ligação tal qual a desejada já ocorre com o controle TBindNavigator, conforme pode ser visto no visualizador de LiveBindings da ferramenta (**Figura 8**), acessível por meio do menu View | LiveBindings Designer. Uma forma alternativa de acesso a este Designer se dá via menu de contexto do próprio componente TBindSourceDB, conforme pode ser visto na **Figura 9**.

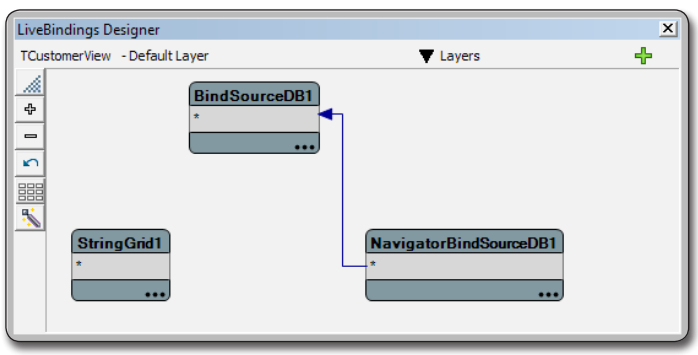


Figura 8. LiveBindings Designer

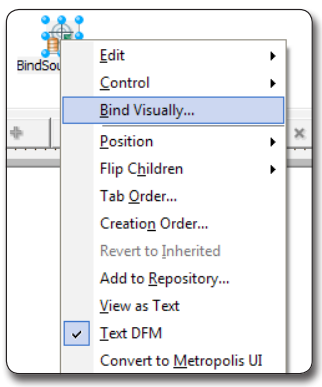


Figura 9. TBindSourceDB – opção Bind Visually

No LiveBindings Designer, o simples arrastar de seta a partir do elemento BindSourceDB1 em direção ao elemento StringGrid1 é o suficiente para que uma ligação entre ambos seja estabelecida. Assim, já com a aplicação em runtime, os dados oriundos da tabela do banco de dados, provindos do modelo, são agora mostrados em tela (**Figura 10**). Por conseguinte, isto vem a comprovar a usabilidade e eficiência do uso do padrão MVVM no desenvolvimento Delphi de aplicações.

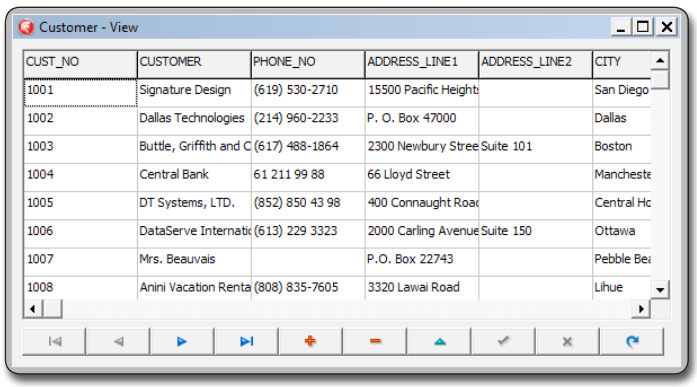


Figura 10. Aplicação MVVM em execução

Com a plena intenção de trazer para a realidade desta comunidade de desenvolvedores mais uma opção concreta de uso, foi apresentado o padrão MVVM e sua introdução e adequação ao cenário Delphi de desenvolvimento. Posto isto, um novo desafio está lançado, que pode ser resumido na astúcia que o leitor deverá ter para tirar proveito dos pontos positivos e de realce que a utilização de padrões deste tipo pode e tende a oferecer às suas construções cotidianas. No final das contas, o grande ganho não se restringe a aplicabilidade de determinada técnica ou abordagem, mas sim pela própria evolução pessoal obtida de forma natural ao se percorrer por novos caminhos até então desconhecidos.

Autor

Fabrício Hissao Kawata
fabricio.kawata@bol.com.br

Formado em Processamento de Dados pela FATEC-TQ e pós-graduado em Engenharia de Componentes. Atua como Analista Programador Delphi há 9 anos. Atualmente escreve artigos sobre Delphi para revistas especializadas. Certified Delphi Developer. Certified Embarcadero Instructor. Site pessoal: www.fabriciokawata.com.br

Links:

RAD Studio XE6 – Página oficial do produto
http://www.embarcadero.com/products/rad-studio

RAD Studio XE6 – Download Trial
https://downloads.embarcadero.com/free/rad_studio

O padrão MVVM
http://msdn.microsoft.com/en-us/library/hh848246.aspx

Você gostou deste artigo?

Dê seu voto em ***www.devmedia.com.br/clubedelphi/feedback***

Ajude-nos a manter a qualidade da revista!

Entenda a Delphi Language

Uma análise sob o aspecto conceitual de linguagens de programação

A história do Delphi começa em 1970 quando foi lançada a linguagem procedural Pascal, que herdou diversas características do ALGOL. Na década de 80 se popularizou através do Turbo Pascal e na década de 90 surgiu o conceito do pascal orientado a objetos (Object Pascal).

Em 1993 a Borland iniciou o desenvolvimento de um projeto para a criação de um ambiente visual para criar aplicações Windows utilizando a linguagem Object Pascal, que foi batizado de Delphi e lançado no ano de 1995.

O arquiteto do projeto e que ficou conhecido como "pai" do Delphi foi Anders Heilsberg (também criador da linguagem C#). Desde o início, Delphi sempre foi pioneiro em tecnologias para o desenvolvimento de softwares, como conexão com banco de dados, programação Orientada a Objetos e ambiente de desenvolvimento rápido (RAD - Rapid Application Development).

A versão 2 do Delphi coincidiu com o Windows 95 e o desenvolvimento para 32 bits, mas como ficamos acostumados, Delphi era totalmente compatível com versão 16 bits. Na versão 3, Delphi inovou mais uma vez, pois foi a primeira ferramenta a ter suporte para criar aplicações multicamadas em um ambiente RAD.

O Delphi 4 trouxe suporte aos mais recentes padrões de mercado naquele momento: CORBA, ORACLE 8, além de criação de controles ActiveX/Active Forms. O Delphi 5 inovou com o suporte a manipulação fácil de arquivos XML, além de suporte a nova tecnologia de acesso a dados: ADO. Além disso, apareceram novas ferramenta como o TeamSource para controle de versão.

A versão 6, trouxe suporte a Web Services, além de BizSnap, DataSnap e WebSnap. Juntamente com a versão 6, nasceu o Kylix, primeira ferramenta RAD para Linux, para criação de aplicativos cross-plataform. No Delphi 7 aparecem os primeiros indícios de suporte a nova plataforma que estava nascendo, o .NET.

Fique por Dentro

Neste artigo veremos porque é tão importante estudar os conceitos relacionados às linguagens de programação. Além disso, iremos fazer uma análise completa da Delphi Language através de uma lista de critérios pelos quais podemos fazer um julgamento da linguagem. Serão feitas análises referentes a Legibilidade, Capacidade de Escrita e Confiabilidade da linguagem, afim de descobrir quais seus pontos fortes e fracos. Por fim, veremos os itens que influenciam no custo de adoção de determinada linguagem para o desenvolvimento de sistemas.

Nessa versão, temos um compilador para .NET, mas o ambiente do Delphi ainda não oferecia suporte, teríamos que criar aplicações totalmente manuais, sem as facilidades de componentes e modelagem rápida. Nesta versão não temos mais o Object Pascal como linguagem, mas sim a Delphi Language, mudança de nome para fins comerciais.

O Delphi 8 suportava apenas aplicações .NET, o que confundiu alguns desenvolvedores e seu IDE mudou drasticamente, sendo mais pesado e consumindo muita memória.

A versão 2005 do Delphi prometia ser a primeira ferramenta a ter ambiente de desenvolvimento para Win32 e .NET (Delphi e C#). Teve início a arquitetura ALM desenvolvida pela Borland, onde temos ferramentas para controlar todo o clico de vida do projeto, desde requisitos até a finalização do mesmo.

O Delphi 2006 foi a última versão da Borland, que juntamente com o Delphi 2007, trouxe algumas novidades no desenvolvimento para .NET e Win32, com a criação de novos componentes para suporte ao Windows Vista e suporte a novas versões do .NET Framework.

No Delphi 2009, já tivemos modificações significativas, onde não existe mais o mesmo ambiente para Win32 e .NET. A CodeGear, então detentora do Delphi, em uma solução radical (no olhar de alguns desenvolvedores), aproveitou que o Visual Studio (principal concorrente) "liberou" o seu ambiente de desenvolvimento

(shell) para outras linguagens e utilizou o mesmo e começou a disponibilizar o Delphi Prism.

Imagine desenvolver em Pascal (na verdade se chama Oxygene, da RemObjects, parceira da Embarcadero), usando o Visual Studio. Foi um verdadeiro choque na comunidade. Na defesa da Embarcadero, a vantagem de usar o ambiente do Visual Studio, seria que o desenvolvedor estaria sempre com a versão mais recente do .NET Framework, além de tecnologias como LINQ, WPF, ASP .NET, Windows Forms e etc.

Mesmo assim, a versão Win32 do Delphi 2009 trouxe novidades como suporte a Generics, Métodos Anônimos e Ribbon. No Delphi 2010, tivemos o suporte a criação de aplicações touchpad, para criar interfaces onde o usuário interage com a mesma usando toque no monitor, uma grande novidade.

Na sequência temos as versões XE e XE2. O XE trouxe suporte ao Windows Azure (desenvolvimento em nuvens) e novidades no DataSnap. O XE2 trouxe novidades significativas como suporte a desenvolvimento de projetos em 64 bits, aplicações multiplataformas para Windows e OSx (Mac), criação de aplicações 3D e HD com um framework chamado FireMonkey e um recurso de LiveBindings para a ligação de dados aos componentes, entre outros.

Na versão XE3 tivemos o lançamento da interface Metropolis UI, para desenvolvermos interfaces gráficas semelhantes às existentes no Windows 8, onde podemos inclusive converter de forma automática interfaces VCL tradicionais para o Metropolis. Também foi evoluído o recurso de LiveBindings, criando uma forma gráfica para fazer a ligação entre componentes de tela e os dados.

Na versão XE4 do Delphi temos o suporte ao desenvolvimento de aplicativos para dispositivos móveis para iOS e no XE5 para Android. Já na versão XE6 temos apenas a melhoria nos recursos existentes e uma remodelagem da interface gráfica do IDE.

Porque estudar os conceitos de uma Linguagem de Programação

É normal que os estudantes se perguntem: por que estudar os conceitos de linguagens de programação? Essa é uma pergunta normal, pois existem vários estudos que se pode fazer de uma linguagem, sua sintaxe, suas bibliotecas nativas, frameworks e plug-ins extras. Mas existe uma série de benefícios que podem ser conseguidos ao estudar os conceitos de uma linguagem de programação, sendo alguns deles os seguintes:

• O aumento da capacidade de codificação de sistemas: conhecer uma variedade maior de recursos de uma linguagem de programação nos possibilita o desenvolvimento de uma gama muito maior de algoritmos e nos permite explorar o que a linguagem tem de melhor para cada escopo de programa. Pode-se abrir a mente para novas formas de implementação de um mesmo problema, trazendo benefícios como um código mais simples e preciso, com um melhor desempenho. O estudo dos conceitos de uma linguagem de programação mostra aos desenvolvedores recursos preciosos que eles poderiam estar desconsiderando até o momento.

• Maior entendimento para escolha da linguagem de programação correta para um problema: muitos desenvolvedores não tiveram uma base sólida conceitual de todos os tipos de linguagens de programação, e muitos aprenderam sozinhos a programar, muitas vezes sem passar por uma instituição de ensino. Treinamentos e cursos específicos de determinada linguagem de programação vão diretamente à sintaxe da linguagem sem oferecer um embasamento teórico que indique que aquela linguagem é mais adequada para determinados tipos de problemas. O fato é que as linguagens de programação não são boas para todos os tipos de projetos de software, sempre existem tipos de projetos onde são mais fracas ou mais fortes. Quando estudamos seus conceitos, conseguimos analisar onde devemos aplicar determinada linguagem e quais os projetos deveríamos considerar a adoção de outra linguagem a fim de facilitar o desenvolvimento e manutenção dos sistemas.

• Capacidade para aprender novas linguagens: ao aprendermos todos os conceitos relacionados a uma linguagem de programação fica muito mais fácil aprender outras linguagens. O mais importante não é aprender as sintaxes de uma linguagem e sim os conceitos relacionados àquela implementação. Considerando um exemplo prático, em sistemas que adotam o paradigma orientado a objetos, temos o mecanismo de herança de classes, neste caso é muito mais importante sabermos como aplicar corretamente a herança na modelagem de um sistema do que a sintaxe específica de determinada linguagem. Quem desenvolve em mais de uma linguagem de programação sabe que é bastante natural a passagem de uma sintaxe para outra, desde que a mesma possua o mesmo paradigma, por exemplo, Delphi e C#.

• Aprender a importância de uma boa codificação: ao conseguir ter uma visão do porquê de determinada linguagem ter sido projetada daquela forma, levando os desenvolvedores a criarem códigos mais inteligentes e claros, de forma que outros desenvolvedores entendam rapidamente qual o objetivo das partes do sistema. Determinados tipos de erros que ocorrem nos sistemas somente são identificados por desenvolvedores mais experientes que conseguem ver além daquela implementação.

• Capacidade de implementar melhores linguagens: é óbvio que muito dificilmente nós desenvolvedores viremos a implementar novas linguagens de programação, mas estudá-las nos possibilita achar os pontos fortes e fracos de determinadas linguagens, de maneira que se fôssemos projetar uma nova linguagem, iríamos procurar aproveitar todos os pontos positivos de uma linguagem e retirar suas falhas, procurando a solução em outras tecnologias.

• O avanço das linguagens de programação: entender o porquê daquela linguagem ter se tornado popular e quais as áreas que elas fazem maior sucesso, fazendo com que sejam aproveitadas nos projetos adequados e dando lugar a outras linguagens melhores nos projetos que não tenham efetividade.

Critérios de Avaliação de Linguagens de Programação

Existem diversos critérios que podem ser adotados a fim de fazer uma avaliação de uma linguagem de programação, dificilmente os cientistas da computação entram em consenso sobre quais são os critérios mais adequados para fazer avaliações da sintaxe das linguagens, porém alguns itens básicos são praticamente um consenso entre quase todos. Estes são os apresentados a seguir,

na **Tabela 1** que mostra em quais critérios as características de uma linguagem se enquadra.

	Critérios		
Característica	**Legibilidade**	**Facilidade de Escrita**	**Confiabilidade**
Simplicidade	X	X	X
Ortogonalidade	X	X	X
Instruções de Controle	X	X	X
Tipos de dados	X	X	X
Projeto da Sintaxe	X	X	X
Suporta a abstração		X	X
Expressividade		X	X
Verificação de tipos			X
Tratamento de exceções			X
Ponteiros Restritos			X

Tabela 1. Critérios de avaliação

Legibilidade

Sem dúvida um dos critérios de avaliação de linguagens de programação mais importantes é o da legibilidade, que é o grau de facilidade com que os algoritmos criados naquela linguagem podem ser lidos e entendidos.

A facilidade de manutenção de aplicações está diretamente ligada à legibilidade do código, por isto essa se torna um dos critérios mais importantes de avaliação.

A legibilidade de um programa também está ligada ao domínio do problema em questão, pois quando determinada linguagem de programação não foi projetada para determinado tipo de problema, é bastante provável que seu código fique confuso, tornando difícil de ser lido e interpretado.

Por exemplo, se quisermos utilizar o Delphi para criarmos aplicações web de grande porte, sabemos que teremos dificuldade, pois o Delphi na sua essência não foi desenvolvido para atender esse nicho de mercado. Agora se quisermos criar aplicações Desktop complexas e com diversos formulários interagindo entre sim, provavelmente teremos uma aplicação simples e clara, pois é onde o Delphi se destaca desde o seu surgimento.

Simplicidade

A simplicidade de uma linguagem de programação está diretamente ligada a sua legibilidade, isto porque uma linguagem com uma sintaxe extensa é muito mais difícil de ser aprendida do que uma linguagem que possua poucos artefatos básicos.

A maioria dos desenvolvedores geralmente aprende apenas parte de uma linguagem de programação, nunca ela no seu todo, ignorando alguns recursos. Este tipo de atitude é tomada principalmente para que não seja necessário um estudo mais profundo da linguagem, tornando o início do desenvolvimento rápido, trazendo resultados também rápidos.

O problema desta abordagem é que outros desenvolvedores podem ter aprendido outra parte da linguagem, diferente daquela que aprendeu o desenvolvedor do sistema, tornando difícil a compreensão do código posteriormente. Um exemplo no mundo Delphi deste problema pode ser visto quando temos alguns desenvolvedores que só sabem trabalhar de maneira procedural, com a adição de componentes e codificação de eventos. Outros desenvolvedores podem ter aprendido trabalhar de forma orientada a objetos, com a instanciação de classes e o seguimento de boas práticas de programação, como princípios SOLID e padrões de projeto.

Um segundo ponto que realmente complica em termos de simplicidade de uma linguagem de programação é o fato de ela ter diversas maneiras de realizar uma única tarefa. Por exemplo, em C para incrementarmos uma variável numérica simples existem diversas maneiras, conforme pode ser visto na **Listagem 1**.

Listagem 1. Incrementando uma variável simples em C

```
1. contador = contador + 1
2. contador += 1
3. contador++
4. ++contador
```

Veja que apesar de em determinadas situações esses operadores terem significados diferentes, quando são aplicados a variáveis simples individualmente eles têm o mesmo propósito, incrementar em um a variável.

Neste ponto o Delphi traz mais simplicidade em relação a outras linguagens, pois existe uma única forma nativa do compilador para incrementar em um uma variável, como mostrado a seguir:

```
1. var contador: integer;
2. contador := contador + 1;
```

Nota

Muitos podem estar pensando que existe sim outra maneira de incrementarmos em um uma variável numérica em Delphi, através do procedure Inc, porém devemos observar que o Inc não é um recurso nativo disponibilizado pelo compilador e sim um procedimento implementado na unit System, o que são coisas totalmente diferentes.

Outra característica que prejudica a simplicidade de uma linguagem de programação é a possibilidade de fazermos sobrecarga de operadores na qual um único símbolo pode ter mais de um significado. Embora seja um recurso útil, pode levar ao desenvolvimento de códigos complexos, principalmente se não for adotado com cautela por parte do criador da sobrecarga.

Um exemplo simples de sobrecarga é a do símbolo de soma (+), onde ele pode tanto somar inteiros ou concatenar strings. Nestes casos é bastante indicado o uso de sobrecarga de operadores, pois torna a linguagem mais simples e enxuta.

Ortogonalidade

A ortogonalidade numa linguagem de programação significa a quantidade pequena de instruções primitivas combinado com

um número pequeno de maneiras para construir estruturas de controle e laços de repetição. Na prática, significa o número de regras que determinada linguagem possui.

Linguagens com baixa ortogonalidade possuem muitas exceções às regras da linguagem, já linguagens com alta ortogonalidade, como o Delphi possuem poucas exceções às regras básicas.

Nota

O recurso de sobrecarga de operadores é suportado no Delphi desde a versão 2005 (somente para .NET) e a partir de 2006 para Win32. Este recurso foi adicionado na época pela Borland, principalmente para manter a compatibilidade com a plataforma .NET.

Como exemplo de regras, podemos definir o bloco begin/end exigido pelo Delphi nas estruturas de controles, rotinas e laços de repetição. Esta é uma regra única de bloco, sendo a única exceção à regra o laço de repetição Repeat, onde não somos obrigados a informar o begin/end, saindo um pouco da regra geral da linguagem.

A ortogonalidade também está diretamente ligada com a simplicidade, quanto mais ortogonal é a linguagem de programação, menos exceções existirão quanto às regras da linguagem. Menos exceções levam a uma codificação mais regular, tornando a linguagem mais fácil de ser lida, entendida e aprendida.

Outro exemplo de exceção à regra existente na Delphi Language é o fato de que todas as instruções obrigatoriamente devem ser finalizadas com ponto e vírgula (;), com a exceção de quando esta instrução estiver dentro de um comando condicional como o IF e for um comando único, conforme pode ser visto na **Listagem 2**.

Listagem 2. Exceção à regra geral no Delphi

```
1. if A <> B then
2.  Writeln('A é diferente de B')
```

Instruções de Controle

Até o final da década de 60 existiam linguagens de programação que tinham muito pouca legibilidade, principalmente em relação às instruções de controle, onde o recurso utilizado era o famoso GOTO para conseguirmos mudar o fluxo de execução de um algoritmo.

No início da década de 70, devido à falta de legibilidade destas linguagens, surgiram iniciativas para resolver estes problemas. Foi aí que surgiram as linguagens de programação estruturadas, principalmente pela falta de legibilidade que o comando Goto causava nos programas da época.

Uma linguagem de programação Top-Down, ou seja, que pode ser lida de cima para baixo, é muito mais fácil de ser entendida do que termos que pular de uma instrução para outra de forma indiscriminada.

A maioria das linguagens de programação projetadas a partir do início da década de 70, devido suas boas estruturas de controle, praticamente eliminou a necessidade de utilização do comando Goto. Apesar de geralmente ele ainda estar disponível na maior parte das linguagens de programação, ele é muito pouco utilizado.

Falando especificamente de Delphi, a maioria dos desenvolvedores nem sabe da existência e funcionamento do comando Goto, mas ele existe na Delphi Language. Na **Listagem 3** podemos visualizar um exemplo simples do funcionamento.

Listagem 3. Utilização do Goto em Delphi

```
1. program ExemploGoto;
2. {$APPTYPE CONSOLE}
3. uses
4.   System.SysUtils;
5. var
6.   Ind: Integer;
7. label
8.   GotoLabel;
9. begin
10.  for Ind := 1 to 5 do
11.  begin
12.    Writeln('Ind = ' + IntToStr(Ind));
13.    if Ind = 3 then
14.      Goto GotoLabel;
15.  end;
16.  Writeln('Final do Loop');
17.  GotoLabel:
18.    Writeln('Final do Loop em ' + IntToStr(Ind));
19.  ReadIn;
20. end.
```

No exemplo temos a definição de um Goto nas linhas 7 e 8, já na linha 14 estamos indicando que queremos saltar para a instrução GotoLabel na linha 17, fazendo com que seja interrompido o laço de repetição.

Tipos de Dados

Um sistema de informações produz resultados através da manipulação de dados. Um item determinante para indicar a facilidade com que eles poderão efetuar tal tarefa diz respeito aos tipos de dados oferecidos pela linguagem utilizada se encaixarem com o domínio do problema real. Por este motivo é muito importante uma linguagem possuir uma gama muito boa de tipos de dados e estruturas.

Os tipos de dados básicos mais comuns da linguagem e que não são criados com base em outros são chamados de tipos primitivos.

O fato de uma linguagem de programação fornecer tipos de dados primitivos adequados para variáveis é outra característica que traz bastante legibilidade. Por exemplo, existem linguagens como C que não possuem o tipo de dado Booleano, sendo necessário os desenvolvedores utilizarem outros tipos de dados. Em linguagens com essa característica geralmente define-se zero para false e um para verdadeiro.

Ao nos depararmos com instruções como estas no meio do código, não temos a clareza do que realmente ela representa dentro

da aplicação, sendo necessário uma análise da documentação do código para termos a convicção de que ela representa uma indicação de verdade.

Além dos tipos de dados primitivos comuns existem outros dois tipos de dados não escalares que são os vetores e os registros.

Projeto da Sintaxe

A sintaxe de uma linguagem de programação tem influência direta na legibilidade de uma aplicação. Existem algumas escolhas na hora de se projetar a linguagem que pode aumentar ou diminuir a legibilidade dos programas:

• Formato dos identificadores: se determinada linguagem restringe a um número muito pequeno de caracteres os identificadores, acaba trazendo uma baixa legibilidade. Algumas linguagens limitam muito o tamanho dos identificadores, de forma que não conseguimos representar de forma adequada principalmente o nome das variáveis. No caso do Delphi, felizmente não temos este tipo de problema, já que podemos ter um número ilimitado de caracteres para representar os identificadores (nomes de variáveis, constantes, funções, etc.).

• Palavras reservadas: a legibilidade de um programa também está altamente atrelada pela forma como estão definidas as palavras reservadas, principalmente as que se referem ao início e fim de um bloco condicional ou iterativo. Algumas linguagens como o Delphi usam um par casado de instruções que indicam o início e fim de um bloco. Isso prejudica a legibilidade dos códigos, pois os blocos de códigos terminam sempre da mesma maneira, tornando difícil às vezes determinar qual o grupo de instruções está sendo finalizado em determinado momento (**Listagem 4**). Algumas linguagens resolvem este problema criando uma sintaxe distinta para cada tipo de bloco, usando, por exemplo, end if e end loop para diferenciar os blocos de código. Outra característica das palavras reservadas é a de se a linguagem de programação permite ou não usá-las para nome de variáveis nos programas. Se permite, pode trazer uma grande confusão, pois muitas vezes não saberíamos se determinada instrução se refere a uma instrução nativa do compilador ou o nome de uma variável. O Delphi neste quesito não deixa a desejar, já que somos impedidos de criar nomes de variáveis com palavras reservadas, ocorrendo erro em tempo de compilação se assim o fizermos.

• Significado das instruções: linguagens de programação devem projetar sua sintaxe de modo que as instruções representem fielmente o que significam e que suas instruções não tenham mais de um significado dentro da construção de um programa. O Delphi neste ponto é uma linguagem bastante clara quanto a sua sintaxe, onde cada instrução representa exatamente seu propósito (if, case, function). Um dos únicos itens em que o Delphi falha neste quesito é no uso da palavra Interface. Ela pode tanto representar a criação de uma Interface na orientação a objetos, quanto pode representar um bloco de interface de declarações de classes e métodos dentro de uma unit, conforme pode ser visto na **Listagem 5** (linhas 2 e 4).

Listagem 4. Falta de legibilidade no final de blocos

```
1. procedure Teste;
2. begin
3.   while True do
4.   begin
5.     if True then
6.     begin
7.       //código
8.     end;
9.   end;
10. end;
```

Listagem 5. Mesma instrução representando duas funções diferentes

```
1. unit Pessoa;
2. interface
3. type
4.   IPessoa = interface
5.     procedure Teste;
6.   end;
7. implementation
8. end.
```

Nota

Alguns plug-ins como o CnPack oferecem um mecanismo de melhoria no entendimento de blocos begin/end aninhados, fazendo uso de cores diferentes para cada tipo de estrutura de controle, aumentando a legibilidade dos códigos e facilitando o entendimento da sintaxe. Na seção **Links** está disponível o endereço para download dessa ferramenta.

Facilidade de Escrita

A facilidade de escrita representa o quão fácil é o desenvolvimento de um programa para um determinado domínio. Isso significa que determinada linguagem de programação pode ter uma facilidade de escrita para um determinado tipo de problema e outras não. É o típico caso onde dizemos que não existe uma linguagem melhor ou pior, tudo depende do tipo de sistema que deseja se deseja desenvolver, cabendo ao desenvolvedor analisar qual é a linguagem mais adequada para cada situação.

Não se pode comparar a facilidade de escrita de duas linguagens de programação escolhendo um domínio específico. Por exemplo, é muito mais fácil criar uma aplicação baseada em interfaces gráficas para Windows Desktop utilizando Delphi do que utilizando a linguagem C e isso não significa que o Delphi seja uma linguagem muito mais fácil de escrita, é que o Delphi foi criado para este fim. O mesmo se pode falar de aplicativos de sistema de baixo nível, usando C a tarefa é bem mais simples de que se fossemos utilizar o Delphi, pois o C foi projetado para atender este fim.

Suporte a Abstração

A abstração é um item fundamental nos atuais projetos de linguagens de programação, pois ela possibilita definirmos estruturas e operações complexas de modo a permitir que muitos dos detalhes sejam ignorados. O nível de abstração permitido por uma linguagem e a naturalidade como ela é descrita tem um

grande impacto na facilidade de escrita. Podemos ter dois tipos de abstrações: processos e dados.

Como exemplo de abstração de processos podemos citar a possibilidade de criarmos classes e métodos para realizarmos a tarefa de exportar determinado conjunto de informações para o Excel. Se não tivéssemos suporte a abstração, esta tarefa teria de ser replicada todas as vezes que fosse necessário realizar esta tarefa, tornando o código mais poluído.

Quanto à abstração de dados, podemos usar como exemplo uma lista de informações de um aluno, como nome, idade e curso. Se utilizássemos uma linguagem que não suporta abstração de dados teríamos que criar três vetores e sincronizá-los de forma que cada posição dos vetores representasse a informação referente àquele aluno. No Delphi temos a possibilidade fazer abstração destes dados utilizando uma estrutura (records) ou uma classe para isso, conforme **Listagem 6**.

Listagem 6. Abstração de dados em Delphi

```
1. Alunos = record
2.   Nome: string;
3.   Idade: Integer;
4.   Curso: string;
5. end;
6.
7. TAluno = class
8. strict private
9.   FNome: string;
10.  FIdade: Integer;
11.  FCurso: string;
12. public
13.   property Nome: string read FNome write FNome;
14.   property Idade: Integer read FIdade write FIdade;
15.   property Curso: string read FCurso write FCurso;
16. end;
```

Expressividade

A expressividade de uma linguagem de programação representa as instruções que sua sintaxe permite de forma a tornar a codificação mais simples e enxuta. Isso se consegue com a adição de operadores que tornam a codificação mais elegante e menos verbosa, simplificando a codificação.

Em algumas linguagens de programação temos a opção de fazer o auto incremento utilizando a instrução cont++, enquanto em outras como o Delphi necessitamos de uma instrução maior (cont := cont + 1), o que torna a linguagem menos expressiva.

Outro recurso das linguagens de programação modernas é o for, que também torna as linguagens mais expressivas, devido ao fato de simplificar bastante a tarefa de criarmos laço de repetição que antes só era permitido através de comandos while.

Confiabilidade

A confiabilidade de um programa se refere à possibilidade da linguagem de programação oferecer alguns recursos que os desenvolvedores possam utilizar, de forma a evitar erros inesperados e bugs nos aplicativos.

Os itens de confiabilidade analisados no artigo serão o da verificação de tipos, tratamento de exceções e restrição de ponteiros na linguagem.

Verificação de Tipos

A verificação de tipos é um recurso existente para garantir que os operandos de um operador sejam de tipos compatíveis. Tipos compatíveis são aqueles que são válidos para o operador, já tipos incompatíveis se referem à aplicação de um operador a um operando de tipo impróprio.

A verificação de tipos é um item muito importante em relação à confiabilidade de uma linguagem de programação. Ela consiste na execução de testes de validação por parte do compilador para detectar erros de atribuição às variáveis.

É muito melhor e mais produtivo quando temos a detecção de erros de atribuição em tempo de compilação do que em tempo de execução, isto porque torna mais rápida a correção e evita que códigos que não tenham sido testados adequadamente acabem indo para ambientes de produção com erros que serão somente detectados posteriormente pelo usuário.

Linguagens que possuem verificação de tipos de forma dinâmica, como o JavaScript, permitem somente a validação dos tipos em tempo de execução. Esta forma de validação por um lado traz maior flexibilidade de desenvolvimento para o programador, mas por outro lado traz uma menos segurança quanto à precisão dos dados.

Nos dias de hoje as linguagens fortemente tipadas possuem uma admiração muito grande por parte dos desenvolvedores de software. Linguagens fortemente tipadas têm seus tipos definidos de forma estática, ou seja, uma determinada variável possui um único tipo fixo especificado antes da compilação.

O Delphi neste sentido atende a este quesito de segurança, pois a verificação de tipos é feita durante a compilação dos programas desenvolvidos. A exceção fica por parte do tipo Variant, onde é permitido fazer a atribuição de qualquer valor para variáveis deste tipo.

Tratamento de Exceções

Grande parte dos hardwares são capazes de detectar erros em tempo de execução, como o estouro de um tipo inteiro. Em linguagens que não são capazes de detectar erro, o que ocorre simplesmente é o encerramento do programa abruptamente e o controle passa para o sistema operacional.

A reação do sistema operacional quando ocorre um erro inesperado e o programa se fecha é a de exibir uma mensagem de erro diagnosticando o problema ocorrido. Tal informação pode ser muito útil ou não fazer nenhum sentido para o usuário do programa.

O recurso de tratamento de exceções é a habilidade que uma linguagem de programação tem de interceptar erros em tempo de execução e tratá-los, continuando com o programa sem que o mesmo precise ser reiniciado.

O tratamento de exceções é um recurso indispensável em qualquer linguagem de programação e o Delphi possibilita este tratamento de modo simples, conforme **Listagem 7**.

Listagem 7. Tratamento de Exceções Delphi

```
1. try
2.   try
3.     Resultado := Num1 / Num2;
4.   except
5.     raise;
6.   end;
7. except
8.   on E: Exception do
9.     Writeln(E.ClassName,':', E.Message);
10. end;
```

Apenas para esclarecimento, na linha 5 temos a instrução Raise, que tem a função de propagar a exceção para o bloco externo, ou seja, ela informa ao compilador que não será ali feito o tratamento de exceção e sim no bloco try/except externo.

Existe um segundo bloco de tratamento de exceção chamado try/finally, que foi criado para a liberação segura de recursos. Todo o código que estiver contido dentro do bloco finally será executado, independente do que ocorrer no interior do bloco try. Na **Listagem 8** temos um exemplo de utilização.

Listagem 8. Uso do Try/Finally no Delphi

```
1. program Exemplo;
2. {$APPTYPE CONSOLE}
3. uses
4.   System.SysUtils, System.Classes;
5. var
6.   Texto: TStringList;
7. begin
8.  Texto := TStringList.Create;
9.  try
10.    Texto.Add('João');
11.    Texto.Add('Batista');
12.    Writeln(Texto.Text);
13.  finally
14.    Texto.Free;
15.  end;
16.  readln;
17. end.
```

No exemplo mostrado temos a liberação do objeto Texto no bloco finally, para que independente do que ocorrer no bloco try, no caso a adição de informações, o objeto seja liberado da memória, evitando a ocorrência de vazamento de memória.

Ponteiros Restritos

Ponteiros são tipos de dados que se referem diretamente a um outro valor alocado em outra posição de memória através de seu endereço. Nos dias atuais a utilização de ponteiros principalmente em linguagens de programação de alto nível é muito rara. Algumas linguagens como o Java nem suportam mais este recurso.

O fato de não suportar o uso de ponteiros, segundo alguns cientistas da computação, torna a linguagem mais segura e confiável.

No Delphi utilizamos o operador ^ (circunflexo) para declarar uma variável do tipo ponteiro. Usamos também o operador @ precedendo variáveis "normais" para obter o endereço delas. O inverso de @ seria usar ^ depois de uma variável-ponteiro, para que possamos acessar o seu valor. Um ponteiro recém declarado que não aponta para nenhum lugar tem o valor nil.

Na **Listagem 9** temos um exemplo simples de emprego de ponteiros na Delphi Language.

Listagem 9. Uso de ponteiros no Delphi

```
1. program Exemplo;
2. {$APPTYPE CONSOLE}
3. uses
4.   System.SysUtils;
5. var
6.   Var1: Integer;
7.   Var2: Integer;
8.   Ponteiro: ^Integer;
9. begin
10.  Var1 := 123;
11.  Var2 := 456;
12.  Writeln('Endereço Var1 = ' + IntToStr(Integer(@Var1)));
13.  Writeln('Valor de Var1 = ' + IntToStr(Var1));
14.  Ponteiro := @Var1;
15.  Writeln('Endereco Ponteiro = ' + IntToStr(Integer(Ponteiro)));
16.  Writeln('Valor do Ponteiro = ' + IntToStr(Integer(Ponteiro^)));
17.  readln;
18. end.
```

Como os ponteiros têm a capacidade de acessar variáveis que não são explicitamente declaradas para ele, há uma possibilidade maior de erros de codificação. Apesar disso, o poder oferecido pelos ponteiros é tão grande que em determinadas tarefas, principalmente aquelas referentes ao desenvolvimento de sistemas básicos, o uso de ponteiros pode se mostrar bastante útil.

Custo da Linguagem

O custo de uma linguagem de programação se refere a muitos pontos que serão elencados nos próximos parágrafos.

O primeiro ponto a ser analisado é o custo que terá o treinamento de desenvolvedores para a linguagem escolhida, que depende muito da simplicidade e da ortogonalidade da sintaxe, além, é claro, do nível atual dos desenvolvedores que serão treinados. Linguagens mais poderosas e com maiores recursos despendem maior investimento de dinheiro e tempo para aprendê-las.

Em segundo lugar temos o custo de desenvolver aplicações para esta linguagem, o que depende da facilidade de escrita, a qual também depende do escopo do problema que deve ser solucionado. Os esforços empregados no desenvolvimento de linguagens de programação de alto nível e IDEs robustos foram feitos para que fosse diminuído o custo de desenvolvimento de softwares.

Tanto o custo de treinar desenvolvedores quanto os de escrever programas na linguagem podem ser minimizados na utilização de um bom ambiente de desenvolvimento, que possibilite a automatização de tarefas básicas e ofereça recursos de facilidades na escrita de código.

O terceiro custo a ser analisado é o de compilar aplicações na linguagem. No passado compiladores demoravam bastante na tarefa de fazer a conversão do código da linguagem no código de máquina que seria executado no hardware. Hoje, felizmente possuímos compiladores rápidos, principalmente também pelo avanço do nível de capacidade dos hardwares.

O quarto ponto é o custo de executar programas escritos em uma linguagem, que está diretamente associado ao projeto dela. Uma linguagem que faz muitas verificações de tipos em tempo de execução, por exemplo, não será rápida, mesmo que o compilador seja muito bom. Hoje em dia o tempo de execução de um aplicativo já não é prioridade no projeto de linguagens, principalmente com os avanços a nível de hardware. No passado, devido principalmente ao baixo nível de memória, os programas deviam ser cuidadosamente escritos e em um nível mais baixo para obter o melhor dos recursos existentes das máquinas limitadas.

Geralmente há de se fazer uma escolha entre o custo de compilação de um programa e a velocidade de execução do mesmo. Existe uma fase da compilação chamada otimização, que é uma técnica utilizada pelos compiladores para diminuir o tamanho e aumentar a velocidade da execução de um determinado programa. Se pouca otimização for feita, a compilação é bem mais rápida, porém a execução do programa se torna mais lenta. Porém, se for despendido bastante tempo para otimização, a compilação será mais demorada, mas teremos um programa executável mais rápido.

O quinto item a ser considerado diz respeito ao custo de utilização da linguagem e/ou IDE de desenvolvimento. Uma linguagem cujas ferramentas de desenvolvimento são caras ou criadas apenas para poucos tipos de hardware terão menores chances de se tornarem populares no mundo do desenvolvimento.

O sexto fator se refere ao custo de uma confiabilidade baixa. Se um sistema crítico falha, como o de controle de voos, por exemplo, o custo de adoção desta linguagem pode se mostra muito alto. Linguagens que possuem um histórico de casos de usos onde houve sérios problemas nos seus sistemas tendem a cair em desuso.

O último fator a ser considerado em relação ao custo é o quando o sistema desenvolvido em determinada linguagem irá depender de manutenções, inclusão de funcionalidades e correção de erros. O custo de manutenção de uma linguagem de programação está

FÓRUM
DEVMEDIA
O lugar perfeito para você ficar por dentro de tudo o que acontece nas tecnologias do mercado atual

No fórum da DevMedia você irá encontrar uma equipe disponível e altamente qualificada com consultores e colaboradores prontos para te ajudar a qualquer hora e sobre qualquer assunto. Temos as salas de Java, .NET, Delphi, Banco de Dados, Engenharia de Software, PHP, Java Script, Web Design, Automação comercial, Ruby on Rails e muito mais!

ACESSE AGORA
www.devmedia.com.br/forum

DEVMEDIA

diretamente ligado à legibilidade da mesma e de um bom desenvolvimento por parte do desenvolvedor.

A facilidade de manutenção é um fator crucial nos dias atuais, pois se calcula que atualmente o custo de manutenção supera em até em quatro vezes o custo do desenvolvimento inicial do sistema.

De todos os fatores que influenciam no custo de desenvolver em determinada linguagem de programação, os mais importantes são: facilidade de desenvolvimento de programas, facilidade na manutenção e confiabilidade no resultado.

Existem muitos outros critérios que podem ser levados em consideração na avaliação do custo, como a portabilidade, que é a capacidade de uma implementação ser movida para outro hardware ou sistema operacional, mas não são fatores tão decisivos como os descritos acima.

A linguagem Delphi possui uma sintaxe poderosa, com recursos avançados, como o uso de estruturas e ponteiros. O IDE foi evoluindo com o passar dos anos e nas últimas versões passou a possibilitar o que existe de mais promissor no mercado, a possibilidade de desenvolvimento de aplicativos para dispositivos móveis e com certeza irá impulsionar a ferramenta e ter maior visibilidade entre os desenvolvedores, aumentando a sua popularidade.

Autor

FILIPE DALEPIANE
filipe.dalepiane@gmail.com
http://filipedalepiane.blogspot.com.br/
Analista de Sistemas numa Universidade.
Colunista das revistas Clube Delphi e .NET Magazine.
Desenvolvedor Delphi e C#, utilizando bancos de dados Firebird e SQL-Server. Bacharel em Ciência da Computação. Certificação Delphi Developer

Links:

Plugin CnPack Delphi
http://www.cnpack.org

Você gostou deste artigo?

Dê seu voto em ***www.devmedia.com.br/clubedelphi/feedback***

Ajude-nos a manter a qualidade da revista!

Sistema de créditos DevMedia

Conteúdo **completo** ao seu alcance

Com o sistema de créditos DevMedia você pode comprar individualmente mais de 4.000 vídeo aulas, 8.000 artigos, além de palestras e vídeos com formações completas.

TODAS as vídeo aulas que você sempre quis ver a partir de

R$ 1,50

ECONOMIA: você não precisa assinar 8 produtos diferentes para ter acesso ao conteúdo completo do site!

Saiba mais sobre o Sistema de Créditos!

www.devmedia.com.br/creditos

DEVMEDIA

Somos tão apaixonados por tecnologia que o nome da empresa

diz tudo.

Porta 80 é o melhor que a Internet pode oferecer para sua empresa.

Já completamos 8 anos e estamos a caminho dos 80, junto com nossos clientes.

Adoramos tecnologia.
Somos uma equipe composta de gente que entende e gosta do que faz,

assim como você.

Estrutura

100% NACIONAL. Servidores de primeira linha, links de alta capacidade.

Suporte diferenciado

Treinamos nossa equipe para fazer mais e melhor. Muito além do esperado.

Serviços

Oferecemos a tecnologia mais moderna, serviços diferenciados e antenados com as suas necessidades.

1-to-1

Conhecemos nossos clientes. Atendemos cada necessidade de forma única.
Conheça!

Porta 80
WEB HOSTING

Hospedagem | Cloud Computing | Dedicados | VoIP | Ecommerce | Aplicações | Streaming | Email corporativo

porta80.com.br | comercial@porta80.com.br | twitter.com/porta80

SP 4063-8616 | RJ 4063-5092 | MG 4063-8120 | DF 4063-7486